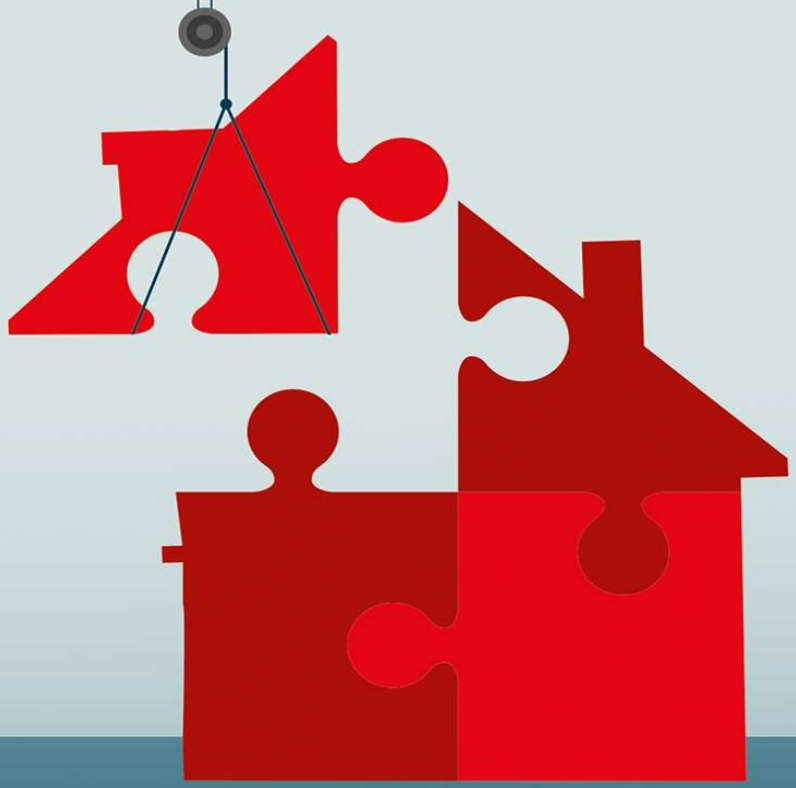
Uma abordagem prática

Introdução à Orientação a Objetos com C++ e Python



novatec

Orlando Saraiva Jr.

Introdução à Orientação a Objetos com C++ e Python

Orlando Saraiva Jr.

Novatec

Copyright © 2017 da Novatec Editora Ltda.
Todos os direitos reservados e protegidos pela Lei 9.610 de 19/02/1998. É proibida a reprodução desta obra, mesmo parcial, por qualquer processo, sem prévia autorização, por escrito, do autor e da Editora.
Editor: Rubens Prates
Revisão gramatical: Marta Almeida de Sá
Editoração eletrônica: Carolina Kuwabata
Capa: Carolina Kuwabata
ISBN: 978-85-7522-549-3
Histórico de edições impressas:
Março/2017 Primeira edição
Novatec Editora Ltda.
Rua Luís Antônio dos Santos 110
02460-000 – São Paulo, SP – Brasil
Tel.: +55 11 2959-6529
E-mail: *novatec@novatec.com.br*
Site: *www.novatec.com.br*
Twitter: *twitter.com/novateceditora*
Facebook: *facebook.com/novatec*
LinkedIn: *linkedin.com/in/novatec*

Para Ana e Lucas, com amor.

Sumário

Agradecimento

Agradeço primeiro a Deus pelas bênçãos que recebo diariamente.

Obrigado, Deus, pelas pessoas as quais tive o prazer de conhecer em minha jornada profissional. Meus amigos e colegas da Unesp, da FHO-Uniararas e toda a equipe da editora Novatec.

Deus me abençoa com meu trabalho, permitindo-me ser útil ao próximo. Agradeço a Ele por poder de alguma forma contribuir com a formação técnica dos meus alunos e com a sua, caro leitor.

Meu muito obrigado!

Sobre o autor

Orlando Saraiva do Nascimento Júnior é mestre em Tecnologia pela Universidade Estadual de Campinas (FT – Unicamp, 2013), fez MBA em Gestão Estratégica de Negócios (Unifian, 2008) e MBA em Gestão Empresarial (FHO – Uniararas, 2016), é graduado em Tecnologia em Informática pela Universidade Estadual de Campinas (CESET – Unicamp, 2005), assistente suporte acadêmico na Universidade Estadual Paulista (UNESP – campus Rio Claro) e docente na Fundação Hermínio Ometto (FHO – Uniararas) no curso Sistemas de Informação.

Prefácio

Na minha época de graduação, dizíamos que programação orientada a objetos é como sexo na adolescência: muita gente fala, pouca gente faz e quem faz, faz malfeito. Aprender este paradigma de programação é mais do que entender os conceitos de classes, objeto, herança e polimorfismo. É pensar o software como pequenas unidades que trocam dados por meio de mensagens.

Allan Kay, um dos pioneiros no uso do termo Orientado a Objetos (OO) e um dos criadores da linguagem SmallTalk, disse: “Pensei em objetos como células biológicas e/ou computadores individuais em uma rede, capazes de comunicar somente por meio de mensagens (mensagens surgiram no início – levou um tempo para ver como fazer mensagens em uma linguagem de programação de forma eficiente, o suficiente para que seja útil).”

E este é um dos desafios em cursos de graduação relacionados à Tecnologia da Informação: como ensinar o aluno a pensar o programa como células biológicas ou computadores em uma rede cuja comunicação ocorre por troca de mensagens? Desde 2009 sou docente no curso Sistemas de Informação e acompanho de perto este desafio.

Neste livro introdutório, faço um convite a você: saia da zona de conforto, saia do modelo sequência-seleção-iteração e pense em um problema de forma OO. Não é vantajoso aplicar os moldes classe e objeto sem aproveitar as reais vantagens de OO.

Para fortalecer seu aprendizado usaremos duas linguagens de programação: C++ e Python. Você pode me questionar: por que não Java, C# ou Ruby? Nosso objetivo neste livro é a introdução à orientação a objetos. Seu aprendizado será potencializado caso aplique os conceitos adquiridos aqui, em outras linguagens OO.

No capítulo 1, iremos bater um papo sobre a linguagem C++. No capítulo 2, vamos conhecer a linguagem Python. No capítulo 3, conversaremos sobre o que é pensar estruturado. Dando sequência, no capítulo 4 e 5 falaremos sobre classes e objetos. O conceito de composição nos espera no capítulo 6, e herança nos aguarda no capítulo 7. Herança múltipla é o assunto do capítulo 8, enquanto classe abstrata e polimorfismo são os assuntos do capítulo 9. Para finalizar esta jornada, um apêndice com exercícios.

O paradigma OO normalmente é apresentado aos alunos de graduação no terceiro semestre, quando estes já têm conhecimento de pelo menos uma linguagem estruturada (C, Pascal, entre outras). Embora não seja um pré-requisito, recomendo que você conheça ao menos uma linguagem de programação para um melhor aproveitamento desta obra.

Good Hacking

CAPÍTULO 1

Um pouco sobre C++

“Uma pessoa medíocre diz. Uma boa pessoa explica. A pessoa superior demonstra. Uma grande pessoa inspira outros a ver por si mesmos.”

Harvey Mackay, executivo

O que iremos aprender?

- Anatomia de um programa em C++
- Alocação de memória com C++
- Passagem de parâmetros

A linguagem C++

A linguagem C++ foi desenvolvida por Bjarne Stroustrup, do Bell Labs, nos anos 1980, com o objetivo de implementar uma versão distribuída do núcleo do sistema operacional Unix. Stroustrup percebeu que a linguagem Simula tinha características interessantes para o desenvolvimento de software, mas era muito lenta. Resolveu utilizar essas características mescladas com as características da linguagem C (rápida e portável). Inicialmente chamada de “C com classes”, a linguagem foi rebatizada em 1984 de C++.

C++ é uma linguagem compilada. Isso significa que suas instruções computacionais (seus códigos-fontes) precisam ser processadas por um compilador. Esse compilador produz um arquivo-objeto, e os diversos arquivos-objetos são combinados, ligados (etapa que chamamos de *linker*), criando-se um executável, conforme ilustrado na figura 1.1.

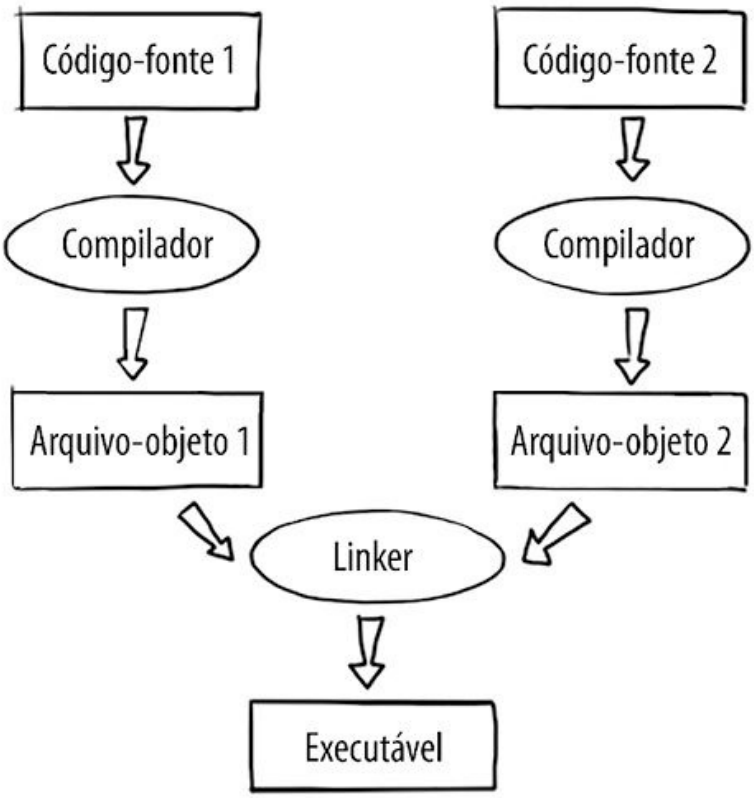


Figura 1.1 – Etapas para criar um executável em linguagens compiladas.

Quando um programa executável é criado para um determinado sistema operacional (como Windows, por exemplo), este não é portável. Caso seja necessário executar em outro sistema operacional (como o Linux), será necessário recompilar o código-fonte, gerando-se um novo executável. Assim, a portabilidade da linguagem está no código-fonte e não no executável gerado.

Anatomia de um programa

O primeiro programa. Sim, sempre ele: o "Oi Mundo". Não se começa a aprender uma nova linguagem sem passar por esta etapa. Vamos ao programa OiMundo em C++:

```
01  /*
02  Programa:   OiMundo
03  Arquivo:    OiMundo.cpp
04  */
05  #include <iostream>
06
07  using namespace std;
08
09  int main() {
10   cout << "Oi Mundo \n";
11   return 0;
12  }
```

Se você já programou em C, sem grandes dificuldades se familiarizará com a sintaxe de C++. A declaração da linha 05 serve para instruir o compilador para que inclua as declarações de entrada e saída presentes nas bibliotecas da linguagem (<iostream>). Sem esta instrução, o programa não irá compilar e a instrução cout na linha 10 não será reconhecida.

Uma dúvida comum é com relação ao uso de <> ou "" para delimitar o nome. Se você utilizar o primeiro conjunto de delimitadores (<>), estará deixando claro para o compilador que o arquivo procurado é uma biblioteca-padrão da linguagem C++. O uso de aspas ("") indica ao compilador que o arquivo a ser inserido está na pasta corrente.

Entre as linhas 9 e 12, observamos a função main (principal). Em C e C++, todo programa começa pela execução dessa função, que retorna ao sistema operacional um valor inteiro. Se nenhum valor for retornado, o sistema operacional entenderá que a execução do programa ocorreu normalmente. Nem todo sistema operacional e nem o ambiente de execução fazem uso deste valor de retorno. Ambientes baseados em Unix-like (Linux, FreeBSD, HP-UX, entre outros) costumam usá-lo. Ambientes baseados em Windows raramente o fazem. Como bons programadores, que respeitam e amam os sistemas operacionais, explicitamente retornaremos um valor 0 ao sistema. No nosso exemplo, na linha 11.

Na linha 10, o operador << ("colocar para") escreve o argumento na saída-padrão. Neste caso, a string literal "Oi Mundo \n" é escrita para o fluxo de saída padrão std::cout. Uma string é uma sequência de caracteres entre aspas duplas. Em uma string, a barra invertida (\) seguida por outro caractere denota um único "caractere especial". Neste caso, \n é o caractere da nova linha, de modo que os caracteres escritos são "Oi mundo!" seguidos por uma nova linha.

Vamos melhorar o programa OiMundo. Agora, queremos receber o nome do usuário e saudá-lo. Criaremos uma função chamada nome para isso (linhas 09-15). Vamos ao segundo exemplo em C++, o programa OiMundo2:

```
01  /*
02  Programa:   OiMundo2
03  Arquivo:    OiMundo2.cpp
04  */
05  #include <iostream>
06
07  using namespace std;
08
09  void nome()
10  {
11   string nome;
12   cout << "Digite seu nome: \n";
13   cin >> nome;
14   cout << "Oi " << nome << ", como vai você? \n\n";
15  }
16
17  int main() {
18   nome();
19   return 0;
20  }
```

Programa Soma

C++ é uma linguagem de tipagem estática, em que cada tipo de cada entidade (objeto, variável, array, matriz) deve ser conhecido pelo compilador. Trabalhar com linguagens estáticas tem suas vantagens, como encontrar erros de tipos antes da execução do código e garantir que os valores possíveis estejam dentro de uma faixa aceitável.

```
01  /*
02  Programa:   Soma
03  Arquivo:    Soma.cpp
04  */
05  #include <iostream>
```

```
06
07  using namespace std;
08
09  float soma(float numero01, float numero02)
10  {
11   cout << "Estamos fazendo soma de floats aqui. ";
12   float resultado;
13   resultado = numero01 + numero02;
14   return resultado;
15  }
16
17  int soma(int numero01, int numero02)
18  {
19   cout << "Estamos fazendo soma de inteiros aqui. ";
20   int resultado;
21   resultado = numero01 + numero02;
22   return resultado;
23  }
24
25  int main() {
26   float numero_float_1 = 5.3;
27   float numero_float_2 = 4.9;
28
cout << "Soma de floats = " << soma(numero_float_1, numero_float_2);
29   int numero_inteiro_1 = 4;
30   int numero_inteiro_2 = 6;
31
cout << "Soma de inteiros = " << soma(numero_inteiro_1, numero_intei
32   return 0;
33  }
```

O programa começa a ser executado na função principal (linha 25). Duas variáveis do tipo float (linhas 26 e 27) e duas variáveis do tipo int (linhas 29 e 30) são alocadas. Ao invocar a função soma (linha 28), é chamada a função soma entre as linhas 9 e 15. Ao ser invocada a função soma novamente (linha 31), executa-se a função entre as linhas 17 e 23. O que determina qual função será chamada são os parâmetros.

A linguagem C++ tem tipos de dados que correspondem a números inteiros, caracteres, valores de ponto flutuante e booleano. Com uso do operador size_of, o programa TamanhoTipos apresenta o tamanho em bytes de cada tipo presente em C++.

```
01  /*
02  Programa:   TamanhoTipos
03  Arquivo:    size.cpp
04  */
05  #include <iostream>
06  using namespace std;
07
08  int main()
09  {
10     cout << "Tamanho do tipo char: " << sizeof(char) << endl;
11     cout << "Tamanho do tipo wchar_t: " << sizeof(wchar_t) << end
12     cout << "Tamanho do tipo string: " << sizeof(string) << endl;
13     cout << "Tamanho do tipo int: " << sizeof(int) << endl;
```

```
14      cout << "Tamanho do tipo short int: " << sizeof(short int) <<
15      cout << "Tamanho do tipo long int: " << sizeof(long int) << e
16      cout << "Tamanho do tipo float: " << sizeof(float) << endl;
17      cout << "Tamanho do tipo double: " << sizeof(double) << endl;
18      cout << "Tamanho do tipo bool: " << sizeof(bool) << endl;
19      return 0;
20  }
```

A saída do programa TamanhoTipos é apresentada a seguir.

```
Tamanho do char: 1
Tamanho do wchar_t: 4
Tamanho do string: 8
Tamanho do int: 4
Tamanho do short int: 2
Tamanho do long int: 8
Tamanho do float: 4
Tamanho do double: 8
Tamanho do bool: 1
```

As variáveis e as memórias

As variáveis são armazenadas na memória. O processo de criar um vínculo, ou seja, associar uma área de memória a uma variável, é chamado de alocação. Quando desvinculamos a região de memória, chamamos de liberação.

Existem quatro categorias de alocação: estática, dinâmica da pilha, dinâmica no heap explícita e dinâmica no heap implícita. As variáveis estáticas são aquelas em que a vinculação ocorre antes do início da execução. Assim, o endereço de memória é vinculado à variável estática e você sabe que ela está lá durante toda a execução do seu programa. Quer usar variáveis estáticas em C e C++? Fácil. Para isso, use a palavra reservada static ao declarar sua variável.

Quando a vinculação é criada a partir de uma sentença de declaração, a chamamos de variável dinâmica da pilha. E adivinha onde esta variável está alocada! Sim, na pilha (stack memory). É alocada e liberada em tempo de execução. Em C++, as variáveis definidas em métodos são, por padrão, dinâmicas da pilha.

As variáveis dinâmicas no heap explícita são variáveis alocadas e liberadas por meio de instruções em tempo de execução. O heap é uma coleção de células de memória sem uma organização ou um padrão de alocação (como ocorre na pilha). Em C++, o operador de alocação é a chamada ao sistema new, e o operador para liberar é a chamada ao sistema delete. As variáveis dinâmicas no heap implícitas são variáveis alocadas no heap apenas quando são atribuídos valores a elas. Variáveis dinâmicas no heap implícitas são comuns em linguagens scripting.

Observe o programa Vinculacao.

```
01  /*
02  Programa:   Vinculacao
03  Arquivo:    Vinculacao.cpp
04  */
05  #include <iostream>
06
07  using namespace std;
08
09  static float numero2 = 6.2;
10
11  void funcao1() {
12   float *ponteiro2 = new float;
```

```
13   *ponteiro2 = numero2;
14
cout << "Endereço do ponteiro na funcao1: " << &ponteiro2 << endl;
15
cout << "Valor do ponteiro na funcao1: " << *ponteiro2 << endl;
16   cout << "Número 2: " << numero2 << endl;
17   numero2 = 4.2;
18   delete ponteiro2;
19
cout << "Valor do ponteiro na funcao1: " << *ponteiro2 << endl;
20  }
21
22  int main() {
23   float numero1 = 10.4;
24   float *ponteiro;
25  /*
26   Em C, seria assim:
27   ponteiro = (float*)malloc(sizeof(float));
28  */
29   ponteiro = new float;
30   *ponteiro = numero1;
31
32   cout << "Número 1: " << numero1 << endl;
33   cout << "Endereço do ponteiro: " << &ponteiro << endl;
34   cout << "Valor do ponteiro: " << *ponteiro << endl;
35
36   numero2 = 5.8;
37   *ponteiro = numero2;
38
39   cout << "Endereço do ponteiro: " << &ponteiro << endl;
40   cout << "Valor do ponteiro: " << *ponteiro << endl;
41
42   funcao1();
43   cout << "Número 2: " << numero2 << endl;
44   delete ponteiro;
45
46   return 0;
47  }
```

Ao executarmos o programa Vinculacao, a saída esperada será algo semelhante à saída a seguir:

```
Número 1: 10.4
Endereço do ponteiro: 0x7ffc7e130720
Valor do ponteiro: 10.4
Endereço do ponteiro: 0x7ffc7e130720
Valor do ponteiro: 5.8
Endereço do ponteiro na funcao1: 0x7ffc7e1306d0
Valor do ponteiro na funcao1: 5.8
Número 2: 5.8
Valor do ponteiro na funcao1: 0
Número 2: 4.2
```

O que aconteceu? Vamos debugar o código. Pararemos a execução do programa em quatro pontos para entendermos o que está acontecendo. O primeiro ponto de parada é a linha 21. A figura 1.2 nos mostra as memórias no Tempo 01. Observe a memória estática, a memória pilha (stack) e a memória heap. A função principal (main) nem começou a rodar, e a variável estática (declarada na linha 7) está lá, alocada.

Um segundo ponto de parada é a execução na linha 31. Este ponto é chamado de Tempo 02 (Figura 1.3). O que mudou do Tempo 01 para o Tempo 02? A função principal (main) foi empilhada (memória stack) e a variável numero1 também. Para a variável numero1 foi atribuído um valor. A variável ponteiro foi alocada na memória heap e está apontando para a variável numero1. Se houver qualquer alteração na variável numero1, muda-se o "valor" da variável ponteiro . Um tipo ponteiro é uma variável que contém um endereço de memória. O tipo ponteiro não tem um valor, ele está apenas apontando para outra variável. Neste caso, ponteiro está apontando para numero1.

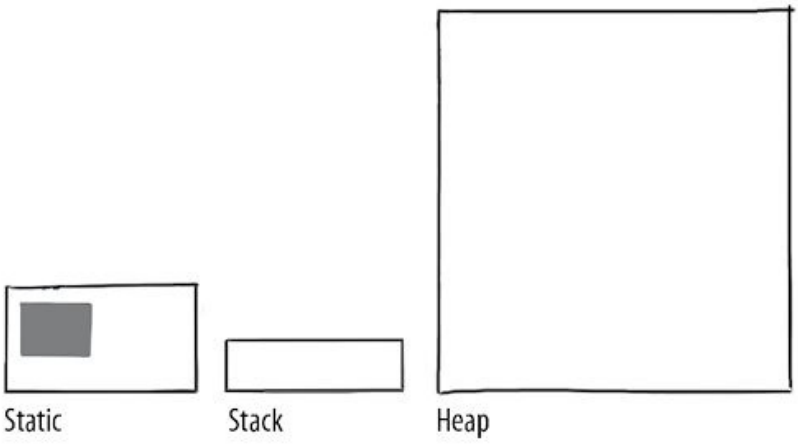


Figura 1.2 – Tempo 01 da execução do programa Vinculacao.

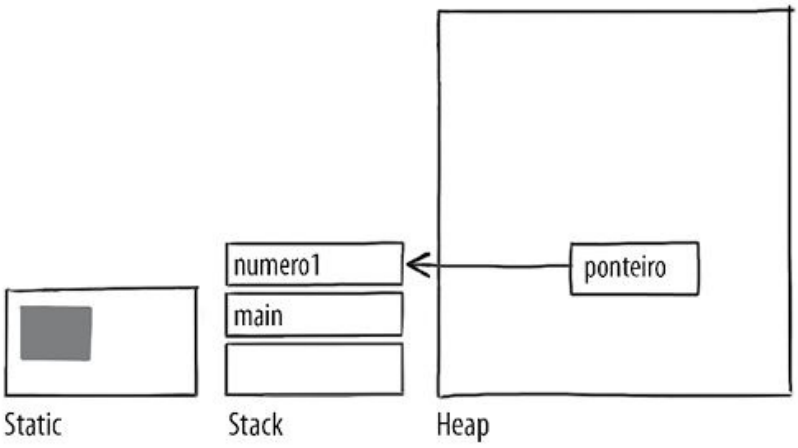


Figura 1.3 – Tempo 02 da execução do programa Vinculacao.

Observe a linha 42. A função funcao1() é invocada e um desvio no fluxo é feito para a linha 9. Entramos dentro da função. empilha-se esta função na memória pilha. Dentro da função há um segundo ponteiro, alocado na memória heap, que está apontando para a memória estática. Observe a figura 1.4, que representa as alocações neste momento. No momento seguinte, o ponteiro2 é liberado com a chamada ao sistema delete e imprime-se o valor do ponteiro. Como esperado, zero, visto que o ponteiro está liberado, e não mais apontando para uma variável.

Um quarto ponto de parada nos aguarda na linha 45. Observe a memória pilha. Apenas a função principal e as variáveis da função principal estão presentes. O ponteiro foi liberado (linha 44). O programa rodou perfeitamente. Nada mais a ser feito, encerra-se a aplicação.

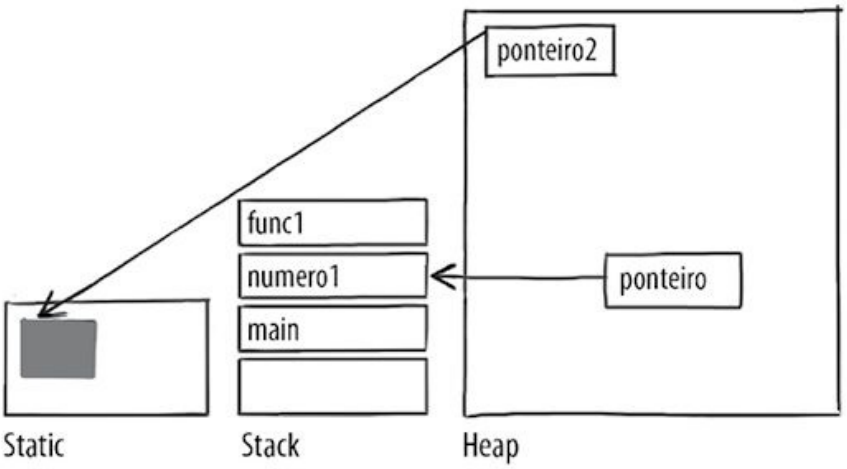


Figura 1.4 – Tempo 03 da execução do programa Vinculacao.

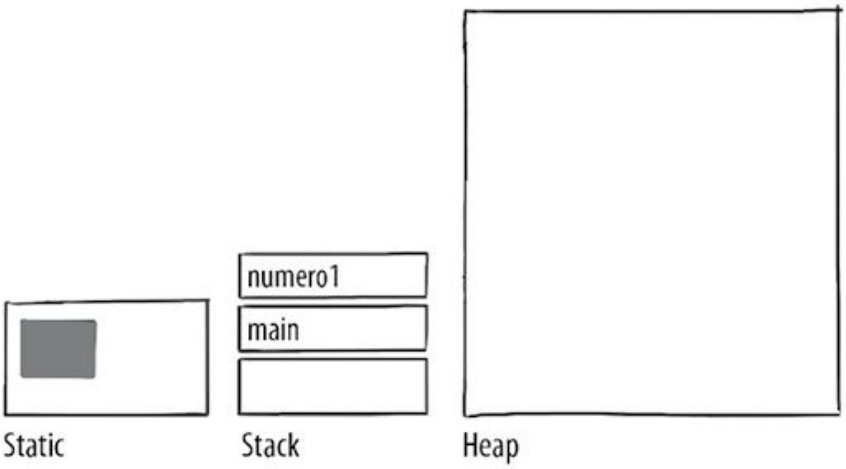


Figura 1.5 – Tempo 04 da execução do programa Vinculacao.

Em linguagens como C++, ao alocar variáveis no heap (new), o programador precisa liberar esta vinculação (delete). Algumas linguagens, como Java, têm mecanismos que limpam estas memórias (*garbage collection*). Para facilitar nossa associação, vamos lembrar de nossas mães. Ao deixar o quarto uma bagunça, como num passe de mágica, ao voltar do colégio, tudo estava limpo e no lugar. Todo mundo tem uma mãe com garbage collection. Mas como dizia a minha: "Você não é todo mundo". Ao retornar do colégio, depois de uma bronca básica, ela me fazia limpar meu quarto.

Passagem de parâmetros

É comum nos códigos estruturados fazer uso de funções. E, para "alimentar" essas funções, precisamos passar dados a elas. Existem dois métodos de passagem de parâmetros geralmente usados: por valor e por referência.

Quando um parâmetro é passado por valor, uma cópia do valor contido é passada à função, e este valor inicia uma variável local. Quando um parâmetro é passado por referência, o endereço de memória em que a variável está é passado para a função. A função tem acesso real ao dado passado, e toda alteração que ocorrer dentro da função irá impactar a variável. A linguagem C++ contém um tipo especial de ponteiro que se chama tipo referência, bem útil para passagem por referência. No código Passagem, há um exemplo didático do uso de passagem por valor e passagem por referência.

```
01  /*
02  Programa:   Passagem
```

```
03  Arquivo:    Passagem.cpp
04  */
05  #include <iostream>
06
07  using namespace std;
08
09  void por_valor(float parametro1,int parametro2)
10  {
11      cout << " Alterando-se valores na função por_valor. ";
12      parametro1 = parametro1 + 10;
13      parametro2 = parametro2 + 5;
14      cout << " parametro1 = "<< parametro1;
15      cout <<" parametro2 = "<<parametro2 << endl;
16  }
17
18  void por_referencia(float &parametro1, int &parametro2)
19  {
20      cout << " Alterando-se valores na função por_referencia. ";
21      parametro1 = parametro1 + 10;
22      parametro2 = parametro2 + 5;
23      cout << " parametro1 = "<< parametro1;
24      cout <<" parametro2 = "<<parametro2 << endl;
25  }
26
27  int main() {
28      float numero_float = 5.3;
29      int numero_inteiro = 4;
30      cout << "numero_float = "<< numero_float;
31      cout <<" numero_inteiro = "<< numero_inteiro << endl;
32      por_valor(numero_float,numero_inteiro);
33
34      cout << "numero_float = "<< numero_float
35      cout <<" numero_inteiro = "<< numero_inteiro << endl;
36      por_referencia(numero_float,numero_inteiro);
37
38      cout << "numero_float = "<< numero_float;
39      cout <<" numero_inteiro = "<< numero_inteiro << endl;
40      return 0;
41  }
```

Ambiente de desenvolvimento

Ambiente de desenvolvimento é o conjunto de ferramentas que o programador utiliza. Pode ser simples: um editor de texto e um compilador apenas. Pode ser uma IDE (Integrated Development Environment – ambiente de desenvolvimento integrado), com diversas ferramentas: compilador, ferramenta de documentação, editor que muda a cor conforme a sintaxe adotada, testes unitários integrados, ferramenta Git para sincronia com o servidor etc.

Como o nosso objetivo é conhecer e explorar orientação a objeto, e não desenvolver sofisticados aplicativos, nosso ambiente de desenvolvimento será simples: o sistema operacional Ubuntu 16.04, editor de texto de sua preferência, terminal shell, compilador g++ versão 5.4.

No decorrer deste livro, quando executarmos um comando no terminal como usuário comum, um símbolo de cifrão ($) aparecerá, conforme exemplo a seguir:

```
$ whoami
orlando
```

Quando for executado um comando no terminal como usuário root, um símbolo de hashtag (#) será mostrado, conforme exemplo a seguir:

```
$ sudo su
# whoami
root
```

Para instalar os pacotes necessários, digite:

```
$ sudo apt-get install build-essential git
```

Após instalar os pacotes necessários, faça o seguinte teste:

```
$ g++ -v
```

Feito este teste, se não ocorrer algum erro informando não ter o pacote instalado, vamos fazer o download dos códigos-fonte que nos acompanhará nos próximos capítulos. Em um diretório de sua preferência, digite:

```
$ git clone https://github.com/orlandosaraivajr/ioo.git
$ cd ioo/
$ ls
01  02  03  04  05  06  07  08  09
```

Cada pasta contém os códigos-fontes dos capítulos deste livro. Estamos prontos para continuar. Para compilar os códigos em C++, utilize o comando make. No exemplo a seguir, os códigos apresentados neste capítulo poderão ser compilados e testados.

```
$ cd 01
$ cd C++
$ make
```

No diretório em que os executáveis estiverem, digite ponto e barra (./) e o nome do programa que desejar rodar.

```
$ ./OiMundo2
```

Para excluir os executáveis gerados, digite o comando make, com o parâmetro clean.

```
$ make clean
```

Síntese

Neste capítulo, apresentamos a linguagem C++ e desenvolvemos alguns programas para exemplificar conceitos da linguagem como tipagem de dados, alocação de memória e passagem de parâmetros. Faremos uso desta linguagem nos próximos capítulos para exemplificar conceitos de orientação a objetos.

CAPÍTULO 2

Um pouco sobre Python

"Que ninguém se engane, só se consegue a simplicidade através de muito trabalho."

CLARICE LISPECTOR (1920-1977), ESCRITORA BRASILEIRA

O que iremos aprender?

- A linguagem Python
- Os tipos presentes em Python
- O ambiente de desenvolvimento

A linguagem Python

Criada por Guido van Rossum nos anos 1990, atualmente a linguagem é mantida nas versões 2.7, 3.4 e 3.5. Multiplataforma, o interpretador Python está disponível para Microsoft Windows e diversos Unix-like.

Python é uma linguagem de alto nível, multiparadigma e interpretada, que pode ser executada em modo script e/ou em modo interativo. A sintaxe de Python é simples e fácil de ser aprendida. O que delimita um bloco na linguagem Python é a indentação, e não o bloco de chaves ({}), como é em C++ ou no uso de uma palavra reservada, como na linguagem Pascal (begin-end).

Quando um programa é criado, o código-fonte é salvo em um arquivo-texto com a extensão apropriada para script Python (*.py*). No terminal, ao invocar o interpretador Python e o script ser criado, o interpretador executa o script e recebe as entradas de dados, conforme é apresentado na figura 2.1.

Figura 2.1 – Execução de linguagens scripting.

Anatomia de um programa Python

O primeiro programa de toda linguagem: "OiMundo". Observe o código.

```
01  #!/usr/bin/python3
02  # coding: utf-8
03  #
04  # Programa: OiMundo
05
06  print("Oi mundo \n")
```

Ao carregar o interpretador, diversas bibliotecas são carregadas. Na linha 1, uma declaração comum para

quem usa Python nos ambientes Unix-like. Se você tiver várias versões do Python instaladas, */usr/bin/env* garante o uso do interpretador desejado. Caso você não declare a linha 1, o interpretador utilizado será o mesmo declarado na variável de ambiente $PATH.

Na linha 2, ocorre uma declaração de qual codificação de arquivo será utilizada. Isso permite o uso de literais Unicode (UTF-8) e faz com que seu código não tenha problemas de acentuação. O uso desta linha faz parte da proposta de melhorias Python (PEP – Python Enhacement Proposal) número 8, a famosa PEP 8. A linha 6 é o nosso programa. A função print recebe uma string literal.

Vamos melhorar o programa OiMundo. Queremos receber o nome do usuário e saudá-lo. E queremos criar uma função chamada nome para isto. Observe o código OiMundo2:

```
01  #!/usr/bin/python3
02  # coding: utf-8
03  #
04  # Programa: OiMundo2
05
06  def nome():
07      nome = input("Digite o seu nome: ")
08      print('Oi ' + nome + ', como está você?\n')
09
10  nome()
```

Entre as linhas 6 e 8, a função nome. A palavra reservada def indica início de uma função. O que limita o escopo da função é a indentação de quatro espaços. Na linha 7, a função input recebe o nome do usuário. É uma função equivalente ao cin do C++. E finalmente, na linha 8, a função print apresenta na tela a mensagem saudando o usuário. Observe como invocar a função nome na linha 10.

Programa Soma

Diferente da linguagem C++, Python é uma linguagem de tipagem dinâmica. O interpretador guarda informações sobre os tipos de entidade. Não é necessário declarar previamente o tipo de entidade em uso.

O uso de linguagens dinâmicas apresenta certas vantagens como flexibilidade no desenvolvimento, mesmo que o preço a se pagar por esta flexibilidade seja uma maior dificuldade em detectar erros, pois a verificação de tipos é feita em tempo de execução. Esta flexibilidade ficará evidente com o código a seguir:

```
01  #!/usr/bin/python3
02  # coding: utf-8
03  #
04  # Programa: soma
05
06  def soma(numero1, numero2):
07      return numero1 + numero2
08
09
10  numero1 = 3
11  numero2 = 4
12  valor = soma(numero1, numero2)
13  print('Soma :' + str(valor) + '\n\n')
14
15  numero2 = 4.2198
16  valor = soma(numero1, numero2)
17  print('Soma :' + str(valor) + '\n\n')
18
19  numero1 = "Oi "
```

```
20  numero2 = "mundo "
21  valor = soma(numero1, numero2)
22  print('Soma :' + str(valor) + '\n\n')
23
24  """
25  Trecho com com erro
26  """
27  numero1 = "Oi "
28  numero2 = 4.2198
29  valor = soma(numero1, numero2)
30  print('Soma :' + str(valor) + '\n\n')
```

Na função soma (linhas 6 e 7), nenhum tipo é previamente definido. Na linha 12, chama-se a função com números inteiros. Sucesso! Na linha 16, chama-se a função com um número inteiro e outro número ponto-flutuante. Sucesso! Vamos invocar a função soma com duas strings na linha 21. Sucesso!

Vamos tentar um parâmetro número ponto-flutuante, e outro, uma string. Aí não dá... Por ser uma linguagem de tipagem forte e dinâmica, o interpretador Python reclama (com razão): TypeError: cannot concatenate 'str' and 'float' objects.

Para conhecermos alguns tipos de Python, vamos utilizá-lo no modo interativo. No terminal, digite **python3**. O interpretador será apresentado da seguinte forma:

```
Python 3.5.2 (default, Jul  5 2016, 12:43:10)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more inform
>>>
```

Na frente dos três sinais de maior (>>>), digite o comando a ser executado e observe a saída.

```
>>> a = 10
>>> a
10
>>> type(a)
<class 'int'>
>>> a = 10.4
>>> type(a)
<class 'float'>
>>> a = 'Oi Mundo'
>>> type(a)
<class 'str'>
>>> a
'Oi Mundo'
```

No primeiro comando, atribuímos à variável a um valor inteiro. Digitando-se a, o interpretador apresenta o resultado. O comando type() nos mostra qual o tipo associado a determinada variável. Atribuímos à variável um valor ponto-flutuante e, em seguida, uma variável string.

O interpretador é responsável por alocar e desalocar as variáveis. Observe o comando id(), responsável por retornar a “identidade” da variável, o número inteiro que é garantido para ser único.

```
>>> a = 10
>>> id(a)
10771808
>>> a = 10.4
>>> id(a)
140250068693472
>>> a = 'Oi Mundo'
```

```
>>> id(a)
140250067508656
>>> a = 10
>>> id(a)
10771808
>>> a = 10
>>> a
10
```

Ao atribuir um inteiro para a variável a e invocar a função id(), obtivemos o número 10771808. Com uma nova atribuição, desta vez com um número ponto-flutuante, outro número é obtido (140250068693472). Com uma nova atribuição, desta vez, com string, um novo número é obtido (140250067508656).

Tipos embutidos

Python contém um conjunto de funções e tipos embutidos (built-in). Tipos embutidos são tipos objetos. Mas, como não nos familiarizamos com o conceito de objeto ainda, trataremos estes tipos embutidos como variáveis que têm funcionalidades embutidas.

Fazendo uso do Python3 em modo interativo, com uso das funções embutidas type(), id() e dir(), vamos explorar algumas destas funções.

```
$ python3
Python 3.5.2 (default, Jul  5 2016, 12:43:10)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more inform
>>> var1 = -10
>>> var2 = 5
>>> type(var1)
<class 'int'>
>>> dir(var1)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__cla
>>> abs(var1)
10
>>> abs(var2)
5
>>> var1 + var2
-5
>>> abs(var1) + abs(var2)
15
>>> help (var1)
```

No código anterior, atribuímos um inteiro negativo à variável var1 e um inteiro à variável var2. Descobrimos o tipo da variável e as funções associadas a esse tipo. Testamos uma dessas funções e, com uso do help, tivemos acesso à documentação do tipo inteiro. Para sair da documentação, pressione a letra q (da palavra quit).

```
>>> var1 = 10.5
>>> type(var1)
<class 'float'>
>>> var2 = 20.6
>>> print(var1)
10.5
>>> print(var2)
20.6
```

```
>>> print(str(var1) + '    '+str(var2))
10.5    20.6
>>> var1 = 5+2J
>>> var2 = 6+1j
>>> print(var1)
(5+2j)
>>> print(var2)
(6+1j)
>>> print(var1 + var2)
(11+3j)
>>> print(str(var1) + '    '+str(var2))
(5+2j)    (6+1j)
>>> type(var1)
<class 'complex'>
>>> var1 = 10.5
>>> print(var1 + var2)
(16.5+1j)
>>> a = True
>>> a
True
>>> type(a)
<class 'bool'>
```

No trecho anterior, exploramos os tipos float, número complexo e booleano. Observe o uso do comando print. Quando imprimimos na tela a soma de dois números, imprime-se o valor resultante. Mas quando o que imprimem-se são os valores, utilizamos a função str(), que serve para converter um tipo para string. Falando em string, vamos explorar este tipo a seguir.

```
>>> var1 = 'Orlando '
>>> var2 = 'Saraiva '
>>> var1 + var2
'Orlando Saraiva '
>>> type(var1)
<class 'str'>
>>> dir(var1)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__
>>> var1.upper()
'ORLANDO '
>>> var1.isalpha()
False
>>> help(var1.isalpha)
>>> var1.isdigit()
False
>>> var1.lower()
'orlando '
>>> var1
'Orlando '
```

O comando help(var1.isalpha) nos mostra uma melhor explicação sobre a função isalpha. Quando se inicia na linguagem Python, o comando help é um recurso importante de aprendizado.

Os tipos string, lista, tupla e dicionários são tipos sequência. String é uma sequência de caractere. Pode-se retornar apenas um subconjunto dessa sequência, conforme podemos observar a seguir:

```
>>> var1
```

```
'Orlando '
>>> var1[2]
'l'
>>> var1[2:5]
'lan'
>>> var1[2:]
'lando '
```

Strings são objetos imutáveis, ou seja, seu estado não pode ser modificado após ter sido criado. Observe como a atribuição de um novo valor à variável var1 gera um novo objeto:

```
>>> var1
'Orlando '
>>> id(var1)
140072712174832
>>> var1 = 'OrLaNdO'
>>> id(var1)
140072712218192
```

Um tipo sequência, porém mutável, é o tipo lista.

```
>>> var1 = [1,2,3,4,5,6,7]
>>> var2 = [8,9,10,'Oi mundo']
>>> type(var1)
<class 'list'>
>>> id(var1)
140195378004104
>>> for i in var2:
...     var1.append(i)
...
>>> id(var1)
140195378004104
>>> var1
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 'Oi mundo']
>>> dir(var1)
['__add__', '__class__', '__contains__', '__delattr__', '__delit
>>> var1.reverse()
>>> var1
['Oi mundo', 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> var1.append([10,11,12,13])
>>> var1
['Oi mundo', 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, [10, 11, 12, 13]]
>>> var1[1]
10
>>> var1[10]
1
>>> var1[11]
[10, 11, 12, 13]
```

Utilizamos a função append() de var1 para adicionar os itens da lista var2 em var1. Observamos a função reverse(). Repare como podemos incluir não apenas elementos de um único tipo. Na lista var1, temos inteiros, string e até mesmo outra lista.

Outro tipo mutável que iremos analisar é o tipo dicionário. Dicionários são mapeamentos, caracterizados por um conjunto de chaves/valor. Observe o exemplo.

```
>>> capitais = {}
>>> capitais = {'São Paulo': 'São Paulo', 'Salvador': 'Bahia', '
>>> capitais
{'São Paulo': 'São Paulo', 'Salvador': 'Bahia', 'Belo Horizonte'
>>> capitais['Salvador']
'Bahia'
>>> capitais['Curitiba']='Paraná'
>>> capitais
{'São Paulo': 'São Paulo', 'Salvador': 'Bahia', 'Belo Horizonte'
>>> capitais['Rio Branco']= ['Acre','AC']
>>> capitais
{'Rio Branco': ['Acre', 'AC'], 'São Paulo': 'São Paulo', 'Salvad
>>> capitais['Rio Branco']
['Acre', 'AC']
>>> capitais['Rio Branco'][0]
'Acre'
```

Neste exemplo, criamos um dicionário de capitais brasileiras. Entretanto, ao cadastrar a capital Rio Branco, o valor não foi uma string, foi uma lista.

O último tipo que veremos é a tupla. Este tipo lembra a lista, mas é um tipo imutável, como string. Vamos aos exemplos:

```
>>> tupla = ('oi','mundo',1,2,3,4)
>>> tupla
('oi', 'mundo', 1, 2, 3, 4)
>>> tupla[0]
'oi'
>>> tupla[1]
'mundo'
>>> dir(tupla)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__
```

A linguagem Python é muito dinâmica. No exemplo a seguir, uma tupla é criada, convertida em uma lista e em seguida para uma string.

```
>>> tupla = ('oi','mundo',1,2,3,4)
>>> type(tupla)
<class 'tuple'>
>>> lista = list(tupla)
>>> lista
['oi', 'mundo', 1, 2, 3, 4]
>>> tupla
('oi', 'mundo', 1, 2, 3, 4)
>>> string = str(lista)
>>> string
"['oi', 'mundo', 1, 2, 3, 4]"
```

Ambiente de desenvolvimento

O ambiente de desenvolvimento utilizado neste livro: o sistema operacional Ubuntu 16.04, o editor de texto de sua preferência, terminal shell e o interpretador Python versão 3.5. Fazendo uso do Ubuntu, não será necessário instalar pacote algum.

```
$ python2
Python 2.7.12 (default, Jul  1 2016, 15:12:24)
```

```
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more inform
>>>
$ python3
Python 3.5.2 (default, Jul  5 2016, 12:43:10)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more inform
>>>
```

Como o Ubuntu tem dois interpretadores Python, no cabeçalho do script é necessário identificar o interpretador correto:

```
#!/usr/bin/python3
# coding: utf-8

print("Oi mundo \n")
```

Digite o conteúdo apresentado em um arquivo (*teste1.py*), dê o privilégio de execução e execute o script conforme mostrado a seguir:

```
$ gedit teste1.py
$ chmod +x teste1.py
$ python teste1.py
Oi mundo
$ ./teste1.py
Oi mundo
```

Síntese

Assim como a linguagem C++, faremos uso desta linguagem nos próximos capítulos. Executamos alguns comandos no modo interativo e scripting para exemplificar conceitos da linguagem como tipagem e sintaxe. Há nítidas diferenças entre essas duas linguagens, e isso será enriquecedor para o nosso objetivo-fim: aprender Orientação a Objetos.

CAPÍTULO 3

Motivações para Orientação a Objetos

“Vemos as coisas por lados diferentes e com olhos diferentes.”

BLAISE PASCAL (1623-1662), MATEMÁTICO E FILÓSOFO FRANCÊS

O que iremos aprender?

- Rever as sentenças que compõem a programação estruturada
- As motivações para Orientação a Objetos

Eu não sei qual foi sua primeira linguagem de programação. A minha foi Pascal. Saudosos e felizes tempos nas aulas de algoritmos em que o exemplo dado era como se troca uma lâmpada.

A vida era simples. Resumidamente, saber programar era realizar atribuições e mesclar sequência, seleção e iteração para resolver um problema. Vez ou outra, criava-se uma função ou um procedimento e pronto, sou um ótimo programador. Aí veio Orientação a Objetos (OO) malvadona, e sem nenhuma cerimônia um caminhão de novos conceitos. E a simplicidade foi-se. Porque inventaram OO? Eu conseguia fazer o que eu precisava de forma estruturada!

Se você pensa assim, este capítulo é para você. Iremos revisitar as principais sentenças estruturadas e irei lhe propor um problema a fim de motivá-lo a enxergar Orientação a Objetos com bons olhos. Orientação a Objetos não é um paradigma criador de problemas; é algo que veio para resolver novos problemas em relação aos quais o paradigma estruturado não mostrou-se tão eficiente.

Sentenças

Iremos revisitar a sentença de atribuição. Com essa sentença, a linguagem nos permite alterar as vinculações de valores às variáveis. Observe o programa Atribuicao:

```
01  /*
02  Programa:   Atribuicao
03  Arquivo:    Atribuicao.cpp
04  */
05  #include <iostream>
06
07  using namespace std;
08
09  int main() {
10   int x,y = 0;
11
cout << "1)  O valor de x: " << x << " e o valor de y: " << y << "\n
12   x = x + 4;
13   y += 4; // Operador de Atribuição
14
cout << "2)  O valor de x: " << x << " e o valor de y: " << y << "\n
15   x = x - 2;
16   y -= 2; // Operador de Atribuição
17
cout << "3)  O valor de x: " << x << " e o valor de y: " << y << "\n
18   x = x * 5;
```

```
19   y *= 5; // Operador de Atribuição
20
cout << "4)  O valor de x: " << x << " e o valor de y: " << y << "\n
21   x = x / 2;
22   y /= 2; // Operador de Atribuição
23
cout << "5)  O valor de x: " << x << " e o valor de y: " << y << "\n
24   x = x % 2;
25   y %= 2; // Operador de Atribuição
26
cout << "6)  O valor de x: " << x << " e o valor de y: " << y << "\n
27
28   ++x; // pré-incremento
29   y++;  // pós-incremento
30
cout << "O valor de x: " << x << " e o valor de y: " << y << "\n";
31   --x; // pré-decremento
32   y--;  // pós-decremento
33
cout << "O valor de x: " << x << " e o valor de y: " << y << "\n";
34
35   int z = 5;
36   // Uso de atribuição condicional
37   x = (z > 10 ? 10 : 20);
38  /*
39   if (z > 10) {
40   x = 10;
41   } else {
42   x = 20;
43   }
44  */
45   z = 15;
46   y = (z > 10 ? 10 : 20);
47
cout << "O valor de x: " << x << " e o valor de y: " << y << "\n";
48   return 0;
49  }
```

Neste programa podemos observar a ação de sentenças de atribuição. Nas linhas 28 e 29, podemos observar operadores de pré-incremento e pós-decremento. Nas linhas 31 e 32, observamos operadores de pré-decremento e pós-decremento. Nas linhas 37 e 46, podemos observar como funciona atribuição condicional. Para melhor entendimento, observe como a linha 37 pode ser substituída pelo trecho apresentado a seguir.

```
    if (z > 10) {
        x = 10;
    } else {
        x = 20;
    }
```

A minha recomendação é: tenha um padrão e compartilhe esse padrão com sua equipe de programadores. Se você estiver inserido em uma equipe de programadores, e cada um utilizar seu próprio padrão, acredite, problemas de legibilidade poderão ocorrer.

Programação estruturada

Programação estrutrada é um paradigma de programação anterior ao surgimento de orientação a objetos. Existem três sentenças que compõem o núcleo da chamada programação estruturada. Uma delas é a sequência, que é a série de etapas realizadas para executar uma tarefa. Você se lembra do seu professor de algoritmo? Do exemplo da lâmpada? Este exemplo é o “Oi Mundo” de quem ensina programação estruturada. Se você não lembra, senta que lá vem a história...

O que eu preciso fazer para trocar uma lâmpada? “Pegue a lâmpada e troque-a”, você responderá. Não.

Há uma série de etapas:

1) Pegar uma escada e colocar abaixo da lâmpada.
2) Subir na escada.
3) Desrosquear a lâmpada até sair do bocal (na minha época, eram lâmpadas incandescentes).
4) Rosquear a nova lâmpada no bocal até ficar bem presa.

E por aí vai... Isto é desenvolver a solução, propor um algoritmo a partir de uma série de tarefas.

As sentenças de iteração (ou laço de repetição) permitem que uma sentença (ou coleção de sentenças) seja executada zero, uma ou várias vezes. As estruturas nesta categoria estão apresentadas na tabela 3.1:

Tabela 3.1 – Tipos de sentenças iterativas

Estrutura	Código exemplo	Comentário

while	while(*condição*) {
	  sentenças;
	}
		condição é uma expressão que retorna verdadeiro ou falso. Enquanto verdadeiras, as sentenças dentro do bloco continuarão a se repetir.

for	for (*cond_inicial*; *condição*; *incremento*) {
	  sentenças;
	}
		cond_inicial é a condição inicial, *condição* é a condição de parada e *incremento* é a variável de controle.

do ... while	do { *sentenças*; }while(*condição*);	A diferença do do... while para o while é onde o mecanismo de controle aparece. Neste caso, após a execução da *sentença*.

Laços internos	while(*condição*) {
	  while(*condição*) {
	  sentenças;
	  }
	  sentenças;
	}
		Um while dentro de outro.

	for (*cond_ini1*; *condição*; *incremento*) {
	  for (*cond_ini2*; *condição*; *incremento*) {
	  sentenças;
	  }
	  sentenças;
	}
		A sentença for aninhada é interessante para ações como leitura/gravação de dados em uma matriz, por exemplo.

	do {
	  sentenças;
	  do {
	  sentenças;
	  }while(*condição*);
	}while(*condição*);
		Um do ... while dentro de outro.

Além disso, podemos ter variações, como um laço for dentro de um do...while ou outros níveis de laços internos. Tudo depende do problema que estamos resolvendo.

E, por fim, as sentenças de seleção. Estas permitem escolher entre dois ou mais caminhos de execução em um programa. As sentenças de seleção englobam duas categorias: de dois caminhos ou múltiplos caminhos. Na tabela 3.2, podemos observar as estruturas de seleção.

Tabela 3.2 – Tipos de sentenças de seleção

Estrutura	Código exemplo	Comentário

se	`if (expressão_booleana) {`
	`  sentença A;`
	`} else {`
	`  sentença B;`
	`}`
		Expressão_booleana retorna verdadeiro ou falso. Se a expressão retornar verdadeiro, realizar *sentença A*. Caso contrário, realizar *sentença B*.

se aninhado	`if(expressão_booleana) {`
	`  if(expressão_booleana) {`
	`  sentença;`
	`  }`
	`}`
		uma sentença se dentro de outra

switch	`switch(expressão) {`
	`  case expressão_constante_1 :`
	`sentença_1;`
	`  break;`
	`  case expressão_constante_2 :`
	`sentença_2;`
	`  default :`
	`sentença_padrao;`
	`}`
		sentença de seleção com múltiplos caminhos.

Assim, conhecendo estas estruturas apresentadas na figura 3.1 (sequência, seleção e iteração), dividindo-se o software em unidades menores (funções e procedimentos), podemos nos considerar programadores que dominam a arte de codificar nas linguagens estruturadas.

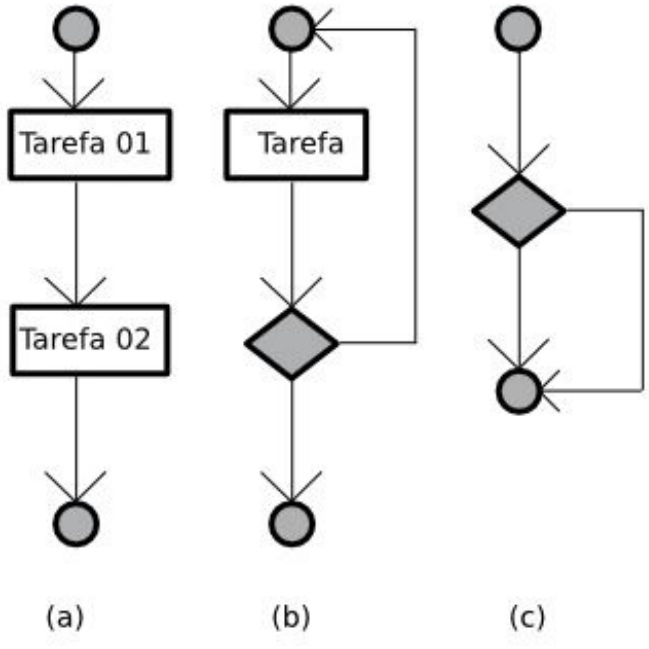


Figura 3.1 – Sentenças da programação estruturada: (a) sequência, (b) iteração, (c) seleção.

Propostas para cadastro de pessoas

Neste momento, quero propor um desafio. Vamos desenvolver um cadastro de três pessoas, com o objetivo de armazenar três informações: nome, idade e sexo. Para os exemplos deste capítulo usarei apenas C++. O número de registros será definido como uma constante com uso do pré-processador define e armazenado em um arquivo chamado *parametros.h*:

```
01  #define N 3
```

Vamos à primeira implementação. Três informações serão armazenadas. Sei o número exato de registros que teremos (definido em *parametros.h*)... Já sei! Três vetores. Um vetor de nomes, um vetor de idades armazenando inteiros e um vetor de sexo, contendo “m” para masculino e “f” para feminino. Vamos à primeira implementação com o programa cadastro01.

```
01  /*
02  Programa:   cadastro01
03  Arquivo:    cadastro01.cpp
04  */
05  #include <iostream>
06  #include "parametros.h"
07
08  using namespace std;
09
10  int main() {
11      int idade[N];
12      string nome[N];
13      char sexo[N];
14      int x; // Variável de controle
15
16      for (x = 0; x < N; x++) {
17          cout << "Digite o nome: ";
18          cin >> nome[x];
19          cout << "Digite a idade: ";
20          cin >> idade[x];
21          cout << "Digite o sexo (m para masculino, f para feminin
22          cin >> sexo[x];
23      }
24
25      for (x = 0; x < N; x++) {
26             cout << nome[x] << "   -   "<< idade[x] << "   -   "
<< sexo[x]<< endl;
27      }
28      return 0;
29  }
```

Certo, certo... Atendeu ao requisito pedido. Este exemplo apenas reforçará sua premissa inicial de que programar estruturado atende bem aos problemas propostos. Mas vamos mudar algumas coisas... Analisando o programa cadastro01, fica uma lacuna. Para lidar com três campos, são necessários três vetores, somando-se a outras variáveis... Depois de alguns anos, como saber o que é informação das pessoas e o que não é? Vamos fazer um upgrade. E se houvesse uma forma de aglomerar em uma mesma unidade o que são informações das pessoas? Você, firme e convicto de que programação estruturada tem todas as ferramentas necessárias, me responde: use estruturas (structs). Concatene informações sobre as pessoas em uma mesma unidade. Observe o programa cadastro02.

```
01  /*
02  Programa:   cadastro02
03  Arquivo:    cadastro02.cpp
```

```
04  */
05  #include <iostream>
06  #include "parametros.h"
07
08  using namespace std;
09  struct Pessoa {
10      string nome;
11      int idade;
12      char sexo;
13  };
14
15  int main() {
16
17      Pessoa cadastro[N];
18      int x; // Variável de controle
19
20      for (x = 0; x < N; x++) {
21          cout << "Digite o nome: ";
22          cin >> cadastro[x].nome;
23          cout << "Digite a idade: ";
24          cin >> cadastro[x].idade;
25          cout << "Digite o sexo (m para masculino, f para feminin
26          cin >> cadastro[x].sexo;
27      }
28
29      for (x = 0; x < N; x++) {
30          cout << cadastro[x].nome << "  -  " << cadastro[x].idade
31          cout << "  -  " << cadastro[x].sexo << endl;
32      }
33      return 0;
34  }
```

O que mudou da primeira para a segunda implementação? Sim! Há uma estrutura que torna coeso o armazenamento das informações solicitadas. Cada variável é um membro de uma estrutura. E se for preciso mais um membro? Basta alterar a estrutura da struct, incluindo-se uma nova variável, e pronto! A estrutura Pessoa armazenará uma nova informação.

E eis que surge um novo requisito em nosso cadastro: como você bem sabe (ou espero que você saiba), mulheres tendem a não gostar de revelar a idade. Especialmente se a idade da mulher tiver dois dígitos, sendo o dígito decimal igual ou maior que três. Assim, respeitando-se este requisito, para as mulheres nesta condição, não iremos revelar a idade. Iremos apresentar uma frase personalizada.

Queremos implementar tal melhoria. Para isso, surge uma ideia bacana: por que não direcionar a impressão do cadastro para uma função? Sim. Assim, ficará mais fácil fazer a manutenção nesta funcionalidade. Vamos ao código do programa cadastro03.

```
01  /*
02  Programa:   cadastro03
03  Arquivo:    cadastro03.cpp
04  */
05  #include <iostream>
06  #include "parametros.h"
07
08  using namespace std;
```

```
09
10  struct Pessoa {
11      string nome;
12      int idade;
13      char sexo;
14  };
15
16  int x; // Variável de controle
17
18
19  /* Função para mostrar os cadastros  */
20  void mostrar(Pessoa *cad) {
21            cout  <<  "-----------------------------------------
" << endl;
22      for (x = 0; x < N; x++) {
23          if (cad[x].sexo == 'F' && cad[x].idade >= 30) {
24              cout << cad[x].nome << " - ";
25              cout << "não é elegante revelar a idade de uma mulhe
<< endl;
26          } else {
27              cout << cad[x].nome << " - ";
28                           cout  <<  cad[x].idade  <<  "  anos  -
" << cad[x].sexo << endl;
29          }
30      }
31            cout  <<  "-----------------------------------------
" << endl;
32  }
33  /* Função principal (main) */
34  int main() {
35
36      Pessoa cadastro[N];
37
38      for (x = 0; x < N; x++) {
39          cout << "Digite o nome: ";
40          cin >> cadastro[x].nome;
41          cout << "Digite a idade: ";
42          cin >> cadastro[x].idade;
43          cout << "Digite o sexo (m para masculino, f para feminin
44          cin >> cadastro[x].sexo;
45      }
46
47      mostrar(cadastro);
48      return 0;
49  }
```

Analisando-se o código de cadastro03, temos como melhoria a função mostrar, que apresenta os registros. E surge o problema de tratar as entradas de dados. Por que não criar uma função específica para ler os dados? Sim, uma função cadastrar, por meio da qual toda validação da entrada dos dados possa ser feita, para evitar erros como caractere minúsculo quando se espera caractere maiúsculo ou receber caractere texto quando se espera um valor inteiro e maior que zero (como o campo idade, por exemplo).

Surge então a quarta implementação do cadastro, o programa cadastro04, observado a seguir.

```
01  /*
02  Programa:   cadastro04
03  Arquivo:    cadastro04.cpp
04  */
05  #include <iostream>
06  #include <limits>
07  #include "parametros.h"
08
09  using namespace std;
10
11  struct Pessoa {
12      string nome;
13      int idade;
14      char sexo;
15  };
16
17  int x; // Variável de controle
18
19  /*  Função para cadastro dos nomes  */
20  void cadastrar(Pessoa *cad) {
21      int inputSexo;
22
23      for (x = 0; x < N; x++) {
24          cout << "Digite o nome: ";
25          cin >> cad[x].nome;
26
27          /* Validação de input: Permitir números inteios e
28             não aceitar números negativos.  */
29          do {
30              cout << "Digite a idade:  ";
31              cin >> cad[x].idade;
32              if(cin.fail()){
33                  cout << "Digite um número inteiro.\n";
34                  cin.clear();
35                  cin.ignore(std::numeric_limits<streamsize>::max(
36                  cad[x].idade = -1; // Forçar permanecer no laço
37              }
38          } while(cad[x].idade < 0);
39
40          /* Validação do Sexo: 1 para Masculino, 2 para Feminino.
41           cout<<"Selecione o sexo: 1) Masculino 2) Feminino\n";
42            cin>> inputSexo;
43            switch (inputSexo) {
44              case 1:
45                  cad[x].sexo = 'M';
46                  break;
47                case 2:
48                  cad[x].sexo = 'F';
49              break;
```

```
50              default:
51                  cad[x].sexo = 'M';
52              break;
53            }
54      }
55  }
56
57  /*     Função para mostrar os cadastros    */
58  void mostrar(Pessoa *cad) {
59      cout << "-------------------------------------" << endl;
60      for (x = 0; x < N; x++) {
61          if (cad[x].sexo == 'F' && cad[x].idade >= 30) {
62              cout << cad[x].nome;
63              cout << "   não é elegante revelar a idade de uma mu
<< endl;
64          } else {
65              cout << cad[x].nome << " - ";
66                          cout  <<  cad[x].idade  <<  "  anos  -  "
<< cad[x].sexo << endl;
67          }
68      }
69      cout << "-------------------------------------" << endl;
70  }
71
72  /*  Função hackeando.  */
73  void hackeando(Pessoa *cad) {
74      cad[1].nome = "haCK3aDo";
75      cad[1].idade = -2;
76      cad[1].sexo = 'b';
77  }
78
79  /*  Função principal (main)  */
80  int main() {
81      Pessoa cadastro[N];
82
83      cadastrar(cadastro);
84      mostrar(cadastro);
85      hackeando(cadastro);
86      mostrar(cadastro);
87
88      return 0;
89  }
```

Entre as linhas 20 e 55, a elaborada função cadastra trata as entradas dos valores. Tal função é invocada na função principal (main) na linha 83. Na linha 84 a função mostrar é invocada e apresenta os dados corretamente. Na linha 86, o método mostrar é invocado e os resultados não serão os mesmos.

```
Digite o nome: João
Digite a idade:  -5
Digite a idade:  a
Digite um número inteiro.
Digite a idade:  35
```

```
Selecione o sexo:   1) Masculino   2) Feminino
1
Digite o nome: Maria
Digite a idade:  34
Selecione o sexo:   1) Masculino   2) Feminino
2
Digite o nome: Ana
Digite a idade:  24
Selecione o sexo:   1) Masculino   2) Feminino
2
-------------------------------------
João - 35 anos - M
Maria    não é elegante revelar a idade de uma mulher.
Ana - 24 anos - F
-------------------------------------
-------------------------------------
João - 35 anos - M
haCK3aDo - -2 anos - b
Ana - 24 anos - F
-------------------------------------
```

Na execução do programa cadastro04, ao cadastrar o primeiro registro, cometemos propositalmente dois erros: cadastrar uma idade negativa e cadastrar um caractere como idade. Graciosamente, a função cadastrar soube lidar com esse tipo de erro. A impressão dos dados ocorreu como deveria na primeira vez, mas algo aconteceu entre as linhas 84 e 86, afetando-se a impressão da segunda vez. Sim, alguma função (mal-intencionada ou não) alterou indevidamente os dados. Por ser um código pequeno, está fácil localizar onde ocorre o mal-entendido. A função hackeando (linhas 73 a 77) fez um estrago em nossos dados.

Imagine isso ocorrendo em um projeto grande, algo em torno de 500 mil linhas de código, faltando dois dias para a entrega do projeto. Nos testes, ocorre algo parecido. Você e sua equipe precisarão revisar todos os códigos e todas as funções. Como seria o clima na sua equipe de projeto? Maravilhoso. Gente gritando, chorando, o desespero se abatendo em toda a equipe, músicos tocando violino ao estilo do filme *Titanic* quando estava afundando, enfim, o caos em sua plena forma.

Um cadastro para três registros, três campos (nome, idade e sexo), quatro versões implementadas, cada uma atendendo a novos requisitos que foram aparecendo. Imagine em projetos reais, com requisitos e mudanças constantes de requisitos. Depois de quatro versões, parece que ficou um vazio. Como solucionar o problema de alterar os dados indevidamente?

E se a estrutura (struct Pessoa) puder ter funções embutidas, formas de proteger os dados internos e permitir somente manipulação de seu conteúdo via funções? E se for necessário reutilizar esta estrutura, criando-se novas funções para validar os tipos derivados ou melhorar as existentes? Se você está se questionando sobre isso, prepare-se para o próximo capítulo.

Síntese

Neste capítulo, relembramos o conceito de atribuição e revisitamos as estruturas básicas que compõem a programação estruturada (sequência, seleção e iteração). Em seguida, elaboramos um cadastro de pessoas utilizando o paradigma estruturado, quando esbarramos em algumas limitações da programação estruturada como: Há formas de proteger as variáveis internas? Como fazer reúso deste software de forma mais eficiente?

CAPÍTULO 4

Classes e objetos

“Não basta adquirir sabedoria; é preciso, além disso, saber utilizá-la.”

CÍCERO (106–43 A.C.), FILÓSOFO

O que iremos aprender?

- Conceitos-chave de orientação a objetos: classe, objeto, atributo e método
- Separar a interface de programação em C++
- Primeiro diagrama UML: diagrama de classe
- Orientação a Objetos com Python

O conceito de programação orientada a objetos não é algo tão novo. Surgiu na linguagem Simula 67. Foi se desenvolvendo durante os anos 60 e 70 até chegarmos à primeira linguagem considerada puramente orientada a objetos: Smalltalk 80. No capítulo anterior, criamos um cadastro para três pessoas. Nesse cadastro foi necessário armazenar três dados: nome, sexo e idade. Para isso, em nossa última implementação de forma estruturada, fizemos uso de estruturas (struct).

Assim como vetores e matrizes, estruturas permitem agregação de diversas variáveis. Trata-se de um tipo abstrato de dado. A diferença básica entre elas: com uso de vetores e matrizes, todas as variáveis são do mesmo tipo e têm o mesmo nome, enquanto estrutura permite a agregação com tipos diferentes. No problema anterior, criamos a estrutura Pessoa, conforme mostra o código a seguir.

```
struct Pessoa {
    string nome;
    int idade;
    char sexo;
};
….
int x;
Pessoa cadastro;
```

Quando nós definimos a estrutura Pessoa, criamos um vetor de estrutura em que cada posição representa uma pessoa cadastrada. Nesta exemplificação do código, a variável cadastro é uma variável do tipo Pessoa. Assim como a variável x é uma variável do tipo inteiro. O que diferencia estas duas variáveis é o tipo. O tipo inteiro (int) já estava definido anteriormente, enquanto o tipo Pessoa é um tipo definido pelo usuário.

E ficou a dúvida: eu poderia ter um tipo abstrato de dado criado pelo programador, permitindo criar funções associadas à estrutura, protegendo as variáveis internas? Agora você pode, usando classes.

A estrutura básica de uma classe e um objeto é apresentada no código a seguir. A classe é definida pela palavra-chave class seguida pelo nome da classe. A palavra reservada private indica quais serão os dados e funções privativos. A palavra public indica quais serão os dados e funções públicos. O fechamento dos parênteses indica o fim da classe ({}).

```
class NomeClasse {
    private:
        atributos e métodos (privados)
    public:
        atributos e métodos (públicos)
}
NomeClasse objeto
```

As classes são estruturas estáticas que permitem ao programador descrever quais são as variáveis internas

(atributos) e quais as funções (métodos) associadas a este tipo abstrato de dado. Em orientação a objetos, é definida a estrutura estática informando quais são as propriedades (atributos) e quais os comportamentos esperados (métodos). Ao fazer uso deste tipo especial, estamos usando objetos.

No programa cadastroOO existe uma classe definida: Pessoa (linhas 11 a 62). Nesta estrutura classe, existem três atributos: nome, idade e sexo (linhas 13, 14 e 15) e dois métodos: cadastrar (linhas 21-53) e mostrar (linhas 55-61).

```
01  /*
02  Programa:   cadastroOO
03  Arquivo:    cadastroOO.cpp
04  */
05  #include <iostream>
06  #include <limits>
07  #include "parametros.h" // Arquivo do capítulo anterior
08
09  using namespace std;
10
11  class Pessoa {
12     private:
13   string nome;
14   int idade;
15   char sexo;
16
17     public:
18   Pessoa(){
19   }
20
21   void cadastrar() {
22   int inputSexo;
23   cout << "Digite o nome: ";
24   cin >> nome;
25
26   /* Validação de input: Permitir números inteiros e
27          não aceitar números negativos.  */
28   do {
29   cout << "Digite a idade:  ";
30   cin >> idade;
31   if(cin.fail()){
32   cout << "Digite um número inteiro.\n";
33   cin.clear();
34   cin.ignore(std::numeric_limits<streamsize>::max(), '\n');
35   idade = -1; // Forçar permanecer no laço do... while
36   }
37   } while(idade < 0);
38
39   /* Validação do Sexo: 1 para Masculino, 2 para Feminino. */
40    cout<<"Selecione o sexo: 1) Masculino 2) Feminino\n";
41     cin>> inputSexo;
42     switch (inputSexo) {
43   case 1:
44   sexo = 'M';
```

```
45       break;
46     case 2:
47   sexo = 'F';
48   break;
49   default:
50   sexo = 'M';
51   break;
52     }
53   }
54
55   void mostrar() {
56   if (sexo == 'F' && idade >= 30) {
57   cout << nome <<
"   não é elegante revelar a idade de uma mulher. "<< endl;
58   } else {
59   cout << nome << " - "<< idade << " anos - "<< sexo << endl;
60   }
61   }
62  };
63
64  /*  Função principal (main)  */
65  int main() {
66   Pessoa cadastro[N];
67   int x;
68   for(x=0; x < N; x++) {
69   cadastro[x].cadastrar();
70   }
71  // cout << cadastro[0].nome;
72   for(x=0; x < N; x++) {
73   cadastro[x].mostrar();
74   }
75   return 0;
76  }
```

No código do programa cadastroOO, há um método diferente: Pessoa (linhas 18 e 19). Quando um objeto é instanciado, seus membros podem ser inicializados com uso desta função. Chamamos este método de construtor. No momento, seu uso está sendo limitado. Mas, no decorrer do livro, você vai entender o quão importante esse método é. Você pode ter achado estranho uma coisa: o construtor não tem um tipo de retorno. Não tem mesmo. Especificar um tipo de retorno ou especificar um valor de retorno para um construtor é um erro de sintaxe.

Quando a função principal (main) é invocada (linha 65), o vetor cadastro é criado (linha 66). Este é um vetor do tipo Pessoa. Um vetor de objetos. Os métodos são invocados por mensagens. Assim, nas linhas 69 e 73 ocorrem mensagens nos *N* objetos presentes no código.

Na classe, podemos alterar o método de acesso. Por enquanto, trabalharemos dois métodos: público e privado. Com o método de acesso privado (private), os dados e métodos deste tipo são acessíveis somente aos membros da classe. Lembra-se de nosso problema no capítulo anterior? Pois é. Agora, nome, idade e sexo não são acessíveis fora da classe. Com o método de acesso público (public), os atributos e métodos são acessíveis fora da classe. Em nosso código, o construtor e os métodos cadastrar() e mostrar() são públicos.

Para testar a eficiência dos métodos de acesso, remova o comentário com duas barras (//) na linha 71 do código cadastroOO.cpp:

```
71      cout << cadastro[0].nome;
```

O compilador C++ não vai gostar disso. Embora a mensagem possa ser diferente de compilador para compilador, ao ler a mensagem de erro, uma coisa vai ficar clara: o código tentou acesso a um atributo privado e, pelas regras, isso não vale. Cartão vermelho para a sua alteração de código: não vai compilar.

Vamos separar a interface de programação

Quando se desenvolve em C++, é uma boa prática cada definição de classe ser colocada em um arquivo-cabeçalho (header) diferente. A implementação propriamente dita da classe em outro arquivo e a implementação que faz uso da classe implementada incluem o header com uso do (#include). O arquivo *cadastro.h* nos é apresentado a seguir:

```
01  /*
02      Programa: cadastro
03      arquivo:  cadastro.h
04  */
05  #ifndef CADASTRO_H
06  #define CADASTRO_H
07
08  using namespace std;
09
10  class Pessoa {
11      private:
12          string nome;
13          int idade;
14          char sexo;
15      public:
16          Pessoa();
17          void cadastrar();
18          void mostrar();
19  };
20  #endif
```

O que observamos no arquivo cabeçalho é o uso das diretivas de pré-processamento #ifndef, #define e #endif, que evitam que o código seja incluído novamente caso CADASTRO_H tenha sido definido em outra parte do código.

Outra prática adotada é usar o nome do arquivo de cabeçalho, substituindo-se ponto por *underline* (_) e usando-se letras maiúsculas. Assim, no arquivo *cadastro.h*, observamos a definição da classe como CADASTRO_H (linhas 05 e 06). A seguir, a implementação da classe cadastro.

```
01  /*
02      Programa: cadastro
03      arquivo:  cadastro.cpp
04  */
05  #include <iostream>
06  #include <limits>
07  #include "cadastro.h"
08
09  using namespace std;
10
11  Pessoa::Pessoa() {
12      }
13
14  void Pessoa::cadastrar() {
```

```
15      int inputSexo;
16      cout << "Digite o nome: ";
17      cin >> nome;
18
19      /* Validação de input: Permitir números inteios e
20      não aceitar números negativos     */
21      do {
22          cout << "Digite a idade:  ";
23          cin >> idade;
24          if(cin.fail()) {
25              cout << "Digite um número inteiro.\n";
26              cin.clear();
27              cin.ignore(std::numeric_limits<streamsize>::max(), '
28              idade = -1; // Forçar permanecer no laço do... while
29          }
30      } while(idade < 0);
31
32      /*  Validação do Sexo: 1 para Masculino, 2 para Feminino  */
33      cout<<"Selecione o sexo:  1) Masculino  2) Feminino\n";
34      cin>> inputSexo;
35      switch (inputSexo) {
36          case 1:
37              sexo = 'M';
38          break;
39          case 2:
40              sexo = 'F';
41          break;
42          default:
43              sexo = 'M';
44          break;
45      }
46  }
47
48  void Pessoa::mostrar() {
49      if (sexo == 'F' && idade >= 30) {
50          cout << nome << "   não é elegante revelar a idade de um
<< endl;
51      } else {
52                  cout  <<  nome  <<  "-"  <<  idade  <<  "  anos-
" << sexo << endl;
53      }
54  }
```

No arquivo *cadastro.cpp*, observamos a inclusão do header na linha 07. Esta inclusão permite ligar a definição de classe com a implementação, presente neste arquivo. Observamos também o uso de um operador de resolução de escopo (::). Assim, os métodos implementados mostra (linhas 48-54), cadastrar (linhas 14-46) e construtor (linhas 11-12) pertencem ao escopo da classe Pessoa.

```
01  /*
02      Programa: cadastro
03      arquivo:  main.cpp
04  */
```

```
05  #include <iostream>
06  #include "parametros.h"
07  #include "cadastro.h"
08
09  using namespace std;
10
11  int main() {
12      Pessoa cadastro[N];
13      int x;
14      for(x=0; x < N; x++) {
15          cadastro[x].cadastrar();
16      }
17
18      for(x=0; x < N; x++) {
19          cadastro[x].mostrar();
20      }
21      return 0;
22  }
```

No arquivo *main.cpp*, observamos o uso da classe Pessoa e a inclusão do header (linha 7). Ao executar a aplicação gerada, o resultado será o mesmo. Entretanto, pensando-se em manutenção de código, esta separação entre implementação da classe, arquivo-cabeçalho e código que faz uso da classe torna a reutilização mais simplificada.

Vamos documentar

Quando se desenvolve um software, acredite: a documentação é importante. Mas como se documenta um software orientado a objetos? Não foram os deuses do Olimpo, mas um processo de tentativa e erro. Alguns métodos de análise e projeto OO:

- CRC (Class Responsability Collaborator, Beck e Cunningham, 1989)
- OOP (Object Oriented Analysis, Coad e Yourdon, 1990)
- Booch (1991)
- OMT (Object Modeling Technique, Rumbaugh, 1991)
- Objectory (Jacobson, 1992)
- Fusion (Coleman, 1994)
- Notação Unified Modeling Language (UML)

Foi-se testando, mas o que a comunidade adotou como padrão foi com a notação UML. Esta linguagem unificada de modelagem nos apresenta diversas representações gráficas para melhor descrever o comportamento de um software orientado a objetos. São diagramas UML:

- Diagramas estruturais
  - Diagrama de classes
  - Diagrama de objetos
  - Diagrama de componentes
  - Diagrama de instalação ou de implantação
  - Diagrama de pacotes
  - Diagrama de estrutura composta
  - Diagrama de perfil

• Diagramas comportamentais
- Diagrama de caso de uso
- Diagrama de transição de estados
- Diagrama de atividade
- Diagrama de interação
- Diagrama de sequência
- Diagrama de interatividade ou de interação
- Diagrama de colaboração ou comunicação
- Diagrama de tempo ou temporal

Dentre os diversos diagramas, vamos dar uma olhada no diagrama de classes. Observe-o na figura 4.1. Quanta elegância e simplicidade! Três caixas sobrepostas representam a nossa classe. Na primeira caixa, o nome da classe. Na segunda caixa, os atributos da classe; e na terceira, os métodos da classe.

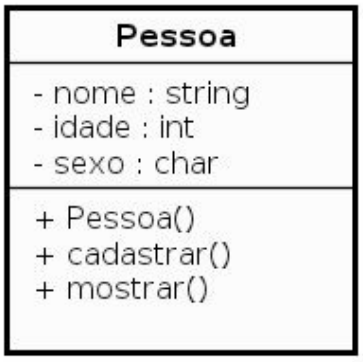


Figura 4.1 – Diagrama de classe da classe Pessoa.

Vamos migrar para Python

Agora que entendemos alguns conceitos-chave de orientação a objetos, queremos migrar para Python. Você já sabe o que é classe, o básico de métodos de acesso (público e privado) e o que é objeto, então vamos migrar!

O código migrado pode ser observado a seguir.

```
01  #!/usr/bin/python3
02  # coding: utf-8
03  #
04  # Programa: cadastro
05  """Cadastro de Pessoa"""
06
07
08  class Pessoa(object):
09      """ Classe Pessoa: responsável em armazenar dados de uma pes
10
11      __nome = ""
12      __idade = -1
13      __sexo = ''
14
15      def __init__(self):
```

```
16          """ Construtor Python """
17
18      def cadastra(self):
19          """ Método cadastra: permite receber os dados de uma pes
20          self.__nome = input("Digite o seu nome: ")
21          while self.__idade < 0:
22              try:
23                  self.__idade = int(input("Digite sua idade: "))
24              except ValueError:
25                  print("Digite número inteiro !")
26          self.__sexo = input("Sexo: M para masculino ou F para fe
27          self.__sexo = self.__sexo.upper()
28          if self.__sexo != 'F':
29              self.__sexo = 'M'
30
31      def mostra(self):
32          """ Método mostra: apresenta os dados cadastrados de uma
33          if self.__sexo == 'F' and self.__idade > 30:
34              print(self.__nome + ' idade secreta  ' + self.__sexo
35          else:
36              print(self.__nome + '' + str(self.__idade) + '' + se
37
38  PESSOAS = list()
39  for i in range(0, 3):
40      OBJ = Pessoa()
41      OBJ.cadastra()
42      PESSOAS.append(OBJ)
43
44  for PESSOA in PESSOAS:
45      PESSOA.mostra()
```

Observe as semelhanças com o nosso código C++:

• O uso da palavra class para identificação de uma classe é um padrão comum em diversas linguagens, não apenas em C++ e Python.

• Uso de construtor. O método __init__ é o construtor de classe em Python.

• Três atributos “privados” e dois métodos.

Está fácil a identificação no código dos atributos (linhas 11 a 13) do construtor (linhas 15 a 16) e dos métodos (linhas 18 a 29 e 31 a 36). A palavra-chave def é a mesma para definir os métodos e as funções. Não se usam as palavras-chave private ou public nos atributos e nos métodos em Python. Usam-se dois underlines (__) para indicar quando um atributo é “privado”. O conceito de privado e público em Python é diferente se comparado ao C++.

Observados as semelhanças e os padrões entre as duas linguagens, vamos agora às diferenças:

• C++ é uma linguagem de tipagem estática. Python é uma linguagem de tipagem dinâmica.

– Em C++, ao declarar os atributos, é necessário associar o tipo ao atributo. O atributo nome é do tipo string. O atributo idade é do tipo inteiro. Em Python, não existe nada disso. Se eu quero atribuir uma string para idade, simplesmente o faço. Sem traumas ou avisos (warning).

• O mesmo para os métodos. Em C++, cada método usa uma assinatura. Uma assinatura é a combinação do nome de um método e seus tipos de parâmetros.

– Em C++, fizemos uso de um vetor de objetos com o número de posições previamente estabelecido. Em Python, conforme mostra a linha 35, fizemos uso de uma lista. E se eu resolver alterar o número de cadastro para quatro e não três, como é hoje? Em C++, altere a variável N e recompile o programa. Em Python, simplesmente use. Crie um novo objeto e use o método append na lista.

• Não existem atributos ou métodos privados em Python

– Não, não existem... Quer dizer que podemos acessar ou modificar um atributo ou método private? Sim, você pode. Este comportamento é cultural da linguagem. Converse com um defensor da linguagem Python. Se você questionar que os atributos não estão protegidos com este comportamento, ele lhe perguntará: “Protegido de quem, de você mesmo?”. Adota-se como boa prática o uso de dois underlines para comunicar que determinado método/atributo é privado, mas nada (apenas o bom senso) lhe impede de mexer no que não se deve.

Python lhe dá grande liberdade. Você pode incluir um novo atributo no objeto durante a execução! Assusta tanta liberdade. Você pode fazer códigos incríveis ou gambiarras que irão trazer grandes problemas futuros, dependendo do seu software e de sua engenharia.

Adicione ao código *cadastro.py* as seguintes linhas:

```
46  print ('Linha 46 ' + str(PESSOAS))
47  print ('Linha 47 ' + str(PESSOAS[0]))
48  print ('Linha 48 ' + str(PESSOAS[0].__dict__))
49  print ('Linha 49 ' + str(PESSOAS[0].__dict__.keys()))
50  print ('Linha 50 ' + str(PESSOAS[0].__dict__.values()))
51  PESSOAS[0]._Pessoa__idade = 'BlaBlaBla'
52  print ('Linha 51  ' + str(PESSOAS[0]._Pessoa__idade))
53  PESSOAS[0]._Pessoa__apelido = 'Cabeção'
54  print ('Linha 52 ' + str(PESSOAS[0].__dict__))
```

Na tabela 4.1, observe o que cada linha faz no programa:

Tabela 4.1 – Alterações no código cadastro.py e as consequências

Linha	Comentário sobre a linha de código
46	O objeto `PESSOAS` é uma lista. Ao imprimir essa lista, observamos que cada item na lista contém um objeto do tipo `Pessoa`
47	Este comando solicita apenas o primeiro item da lista `Pessoas`. Assim, mostra-se o objeto `Pessoa` do primeiro item da lista
48	Os atributos de um objeto ficam armazenados em um dicionário. Com este comando, observamos os três atributos presentes e seus respectivos valores
49	Apresenta apenas as chaves do dicionário que compõem os atributos do objeto
50	Apresenta os valores do dicionário que compõem os atributos do objeto
51	Este comando desnuda toda e qualquer possibilidade de encapsulamento. Estamos diretamente intervindo no primeiro objeto da lista, alterando-se o atributo `idade` com string
52	Aqui, a alteração feita na linha anterior é apresentada. Alteramos um atributo "privado" de um objeto
53	Aqui, um novo atributo é inserido no objeto `Pessoa` em tempo de execução: apelido
54	O dicionário de atributos com o atributo extra

Síntese

Neste capítulo conhecemos a definição de classe e objeto. Entendemos o conceito de atributo e método. Observamos a separação da implementação dos arquivos-cabeçalho, funcionalidade que será muito útil quando criarmos softwares maiores em C++. Analisamos as semelhanças e diferenças entre C++ e Python quando se trata de programação orientada a objetos.

CAPÍTULO 5

Construtores, destrutores e atributos de classe

“É fazendo que se aprende a fazer aquilo que se deve aprender a fazer.”

ARISTÓTELES (384 A.C.-322 A.C.), FILÓSOFO GREGO

O que iremos aprender?

- Construtores e destrutores
- Sobrecarga de métodos
- Atributos de classe, métodos de classe

Abstração é um dos conceitos mais importantes que um programador precisa saber e aplicar. Abstração é uma representação de uma entidade, por meio da qual associamos apenas as características realmente importantes.

Vamos a um exemplo real: queremos comprar um carro. Quais as informações realmente relevantes ao se comprar um carro? Qual a marca? Qual a cor que queremos? Qual o preço do carro no mercado? Estamos interessados no conjunto de dados (atributos) e comportamentos (métodos) que irão representar um carro de forma que os objetos nos represente características que esperamos saber de um carro.

Surge-nos, então, uma pergunta: Se a classe representa os atributos e métodos de um objeto, o que deve ficar para “fora” da classe? Meios de leitura de dados devem ser tratados dentro da classe? Um método como o cadastrar da classe Pessoa do capítulo anterior é considerado um absurdo para alguns defensores de orientação a objetos.

Codificando um carro em C++

A partir da classe Carro, iremos criar alguns automóveis (objetos da classe Carro). Dos diversos atributos que um carro pode ter, os que consideramos mais importantes para o nosso código foram: modelo, cor, ano e preço.

Resumidamente, os comportamentos esperados do nosso carro são:

- apresentar o atributo
- alterar o atributo
- apresentação do carro

Uma série de atributos ficou de fora, como: pagou IPVA ou não, tem som automotivo ou não, qual a quilometragem atual, dentre (muitos) outros. Mas, por se tratar de uma representação simples, ficaremos satisfeitos com esses atributos. O que precisamos ter em mente é que classes são modelagens, representações de objetos concretos ou não. Embora o carro tenha muitos atributos, os que importam para o nosso contexto são os que selecionamos.

Elaboramos o diagrama de classe conforme apresenta a figura 5.1.

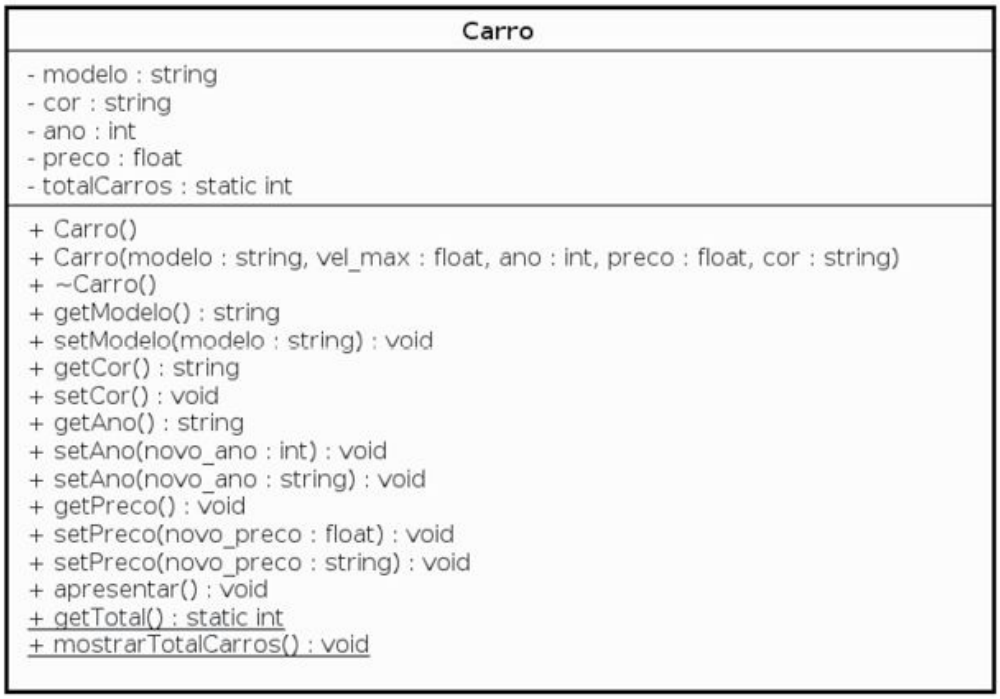


Figura 5.1 – Diagrama de classe.

A partir do diagrama, a codificação em C++ está presente nos arquivos *carro.cpp*, *carro.h* e *main.cpp*.

```
01  /*
02  Programa:   Carro
03  Arquivo:    carro.h
04  */
05  #ifndef CARRO_H
06  #define CARRO_H
07
08  using namespace std;
09
10  class Carro {
11      private:
12          string modelo;
13          string cor;
14          int ano;
15          float preco;
16          /*  Atributo da Classe  */
17          static int totalCarros;
18
19      public:
20          Carro();
21          Carro(string,int,float,string);
22          ~Carro();
23
24          string getModelo();
25          void setModelo(string);
26          string getCor();
```

```
27          void setCor(string);
28          int getAno();
29          void setAno(int);
30          void setAno(string);
31          float getPreco();
32          void setPreco(float);
33          void setPreco(string);
34          void apresentacao();
35
36          /* Método da classe */
37          static int getTotal() {
38              return totalCarros;
39          }
40          static void mostrarTotalCarros() {
41              cout << "Total de carros: " << Carro::getTotal() <<
42          }
43  };
44
45  #endif
```

O arquivo carro.cpp do programa carro é apresentado a seguir.

```
01  /*
02  Programa:   Carro
03  Arquivo:    carro.cpp
04  */
05  #include <iostream>
06  #include <limits>
07  #include <exception>
08  #include "carro.h"
09
10  using namespace std;
11
12  Carro::Carro(){
13      this->modelo = "Fusca";
14      this->ano = 1981;
15      this->preco = 5000;
16      this->cor = "Azul";
17      totalCarros++;
18      cout << "Método construtor chamado. ";
19      cout << "Nenhum modelo configurado, este será um Fusca" << e
20  }
21
22  Carro::Carro(string modelo, int ano, float preco, string cor){
23      this->modelo = modelo;
24      this->ano = ano;
25      this->preco = preco;
26      this->cor = cor;
27      totalCarros++;
28           cout  <<  "Método  construtor  do  carro  "  <<  this-
>modelo << endl;
```

```
29  }
30
31  Carro::~Carro() {
32      totalCarros--;
33       cout << "Método destrutor chamado para o carro " << this-
>modelo << endl;
34  }
35
36  string Carro::getModelo() {
37      return this->modelo;
38  }
39
40  void Carro::setModelo(string novo_modelo) {
41      this->modelo = novo_modelo;
42  }
43
44  string Carro::getCor() {
45      return this->cor;
46  }
47
48  void Carro::setCor(string nova_cor) {
49      this->cor = nova_cor;
50  }
51
52  int Carro::getAno() {
53      return this->ano;
54  }
55
56  void Carro::setAno(string novo_ano) {
57      try {
58          this->setAno(stoi(novo_ano));
59      } catch (exception e) {}
60  }
61
62  void Carro::setAno(int novo_ano) {
63      this->ano = novo_ano;
64  }
65
66  float Carro::getPreco() {
67      return this->preco;
68  }
69
70  void Carro::setPreco(string novo_preco) {
71      try {
72          this->setPreco(stof(novo_preco));
73      } catch (exception e) {}
74  }
75
76  void Carro::setPreco(float novo_preco) {
77      this->preco = novo_preco;
```

```
78  }
79
80  void Carro::apresentacao(){
81      cout << "Carro: " << this->getModelo();
82      cout << " | Cor: " << this->getCor();
83      cout << " | Ano: " << this->getAno();
84      cout << " | Preço: " << this->getPreco() << endl;
85  }
```

O código *main.cpp* do programa carro é apresentado a seguir.

```
01  /*
02  Programa:   Carro
03  Arquivo:    main.cpp
04  */
05  #include <iostream>
06  #include "carro.h"
07
08  using namespace std;
09
10  int Carro::totalCarros=0;
11  int N = 10;
12
13  int main() {
14      Carro *carros[N];
15      Carro carro_estatico1;
16      Carro carro_estatico2;
17
18      carro_estatico1.apresentacao();
19
20      carro_estatico2.setModelo("brasilia");
21      carro_estatico2.setCor("amarelo");
22      carro_estatico2.setAno("1977");
23      carro_estatico2.setPreco("6500");
24
25      cout << "Carro estático 2: " << carro_estatico2.getModelo()
26      carro_estatico2.apresentacao();
27
28      Carro::mostrarTotalCarros();
29
30      carros[0] = new Carro("Duster",2016,25000,"prata");
31      carros[1] = new Carro("BMW",2015,27000,"branco");
32
33      carros[0]->apresentacao();
34      carros[1]->apresentacao();
35
36      Carro::mostrarTotalCarros();
37
38      delete carros[0];
39      delete carros[1];
40
```

```
41      cout << "Total de carros: " << Carro::getTotal() << endl;
42      carros[0] = new Carro();
43      carros[0]->setModelo("Fuscão");
44      carros[0]->setAno(1985);
45      carros[1] = new Carro("Ford Ka",2014,27000,"preto");
46      carros[2] = new Carro("Fiesta",2010,27000,"preto");
47
48      Carro::mostrarTotalCarros();
49
50      carros[0]->apresentacao();
51      carros[1]->apresentacao();
52
53      delete carros[0];
54      delete carros[1];
55      delete carros[2];
56      return 0;
57  }
```

A execução do programa carro é apresentada a seguir:

```
Método construtor chamado. Nenhum modelo configurado, este será
Método construtor chamado. Nenhum modelo configurado, este será
Carro: Fusca | Cor: Azul | Ano: 1981 | Preço: 5000
Carro estático 2: brasilia
Carro: brasilia | Cor: amarelo | Ano: 1977 | Preço: 6500
Total de carros: 2
Método construtor do carro Duster
Método construtor do carro BMW
Carro: Duster | Cor: prata | Ano: 2016 | Preço: 25000
Carro: BMW | Cor: branco | Ano: 2015 | Preço: 27000
Total de carros: 4
Método destrutor chamado para o carro Duster
Método destrutor chamado para o carro BMW
Total de carros: 2
Método construtor chamado. Nenhum modelo configurado, este será
Método construtor do carro Ford Ka
Método construtor do carro Fiesta
Total de carros: 5
Carro: Fuscão | Cor: Azul | Ano: 1985 | Preço: 5000
Carro: Ford Ka | Cor: preto | Ano: 2014 | Preço: 27000
Método destrutor chamado para o carro Fuscão
Método destrutor chamado para o carro Ford Ka
Método destrutor chamado para o carro Fiesta
Método destrutor chamado para o carro brasilia
Método destrutor chamado para o carro Fusca
```

Sobrecarga de métodos

Você reparou que alguns métodos estão “duplicados”? Em linguagens de tipagem estática, quando se tem dois (ou mais) métodos com o mesmo nome, porém os tipos dos parâmetros são diferentes, estamos diante de uma sobrecarga de método.

A explicação é simples, mas por que não refinar a definição a fim de mostrar mais domínio do assunto?

Quando lhe perguntarem a respeito de sobrecarga, responda: “Quando determinada classe tem dois métodos com o mesmo nome, mas com assinaturas diferentes.”

A motivação de sobrecarga é simples. Vamos a um exemplo: na classe Carro, o atributo ano é do tipo inteiro. Mas a classe quer ser legal, oferecendo a possibilidade de o usuário-programador atribuir o ano de duas formas: passando-se um parâmetro tipo string (arquivo *carro.cpp*, linhas 56-60) ou passando-se um parâmetro inteiro (arquivo *carro.cpp*, linhas 62-64). O usuário-programador fica feliz tendo a possibilidade de fazer uso do método de duas formas.

Construtores e destrutores

Quando um objeto é instanciado, os atributos podem ser inicializados pelo método construtor da classe. Construtores podem ser sobrecarregados para oferecer várias formas de instanciar os objetos.

No arquivo *carro.cpp*, fica claro que temos duas formas de executar o construtor. Nesta sobrecarga explícita, há duas formas de criar um objeto: passando-se os parâmetros modelo, ano, preço e cor (arquivo *carro.cpp*, linhas 22-29). Caso queira, poderá construir um objeto chamando o construtor sem parâmetros (arquivo *carro.cpp*, linhas 12-20). Neste caso, o carro será um Fusca, ano 1981, cor azul e com preço de R$ 5.000.

O destrutor de uma classe é um método especial, chamado quando o objeto é destruído. Na verdade, ele (o método) não destrói o objeto, mas faz uma “faxina” ao detectar que o objeto será destruído. Em C++, o destrutor é composto de til (~) e do nome da classe. Tanto o construtor quanto o destrutor não recebem argumento algum e não retornam valor algum.

Atributos de classe e métodos de classe

As classes podem ter dois tipos de métodos e dois tipos de atributos. Os métodos de instância e os atributos de instância são os mais comuns. São os métodos e atributos como os conhecemos até aqui. Você declara a classe, declara os atributos relevantes e os métodos conforme comportamentos esperados. O que difere dois objetos do mesmo tipo são os valores de seus atributos. Cada objeto contém atributos conforme definido na classe. Já os atributos da classe não pertencem às instâncias (objetos), mas sim à classe. Existe apenas uma cópia deste atributo, pertencente à classe somente. O segundo tipo de método, o método de classe, pode realizar computação na classe e nos objetos.

Em nosso código, podemos observar nitidamente quem é o atributo da classe: totalCarros (arquivo *carro.h*, linha 17). Nesta variável do tipo inteiro, cabe a responsabilidade de saber quantos carros existem criados atualmente. A cada objeto criado, o construtor incrementa um valor. A cada objeto destruído, decrementa-se um valor. O método de classe também se faz presente no arquivo *carro.h*: o método getTotal (linhas 37-39) e o método mostrarTotalCarros (linhas 40-42). A forma como o método é invocado deixa claro que este não pertence a algum objeto específico (arquivo *main.cpp*, linhas 28, 36 e 48).

Alocação na pilha e no heap

A alocação na pilha é a alocação do objeto na memória de pilha (stack). Os objetos são criados na pilha e ficam disponíveis para uso durante todo o escopo.

A alocação no heap de memória possibilita alocar um objeto na memória heap (com o comando new) atribuindo o endereço da memória alocada a um ponteiro e fazendo uso desse ponteiro para manipular a memória. Ao terminar de usar o objeto, desaloca-se a memória (com o comando delete). Quando um objeto sai do escopo, ele executa a operação delete sobre os atributos do ponteiro.

Calma, vai ficar simples. Observe o código nos trechos em que ocorre alocação na pilha. No arquivo *main.cpp*, nas linhas 15 e 16, dois objetos são criados. O construtor sem parâmetros é executado, e o primeiro carro é um fusca 1981. Em toda a execução do código, os objetos na pilha estão lá, prontos para ser usados. Observe que o destrutor é invocado apenas ao término do programa. O método da classe getTotal é invocado, informando-nos que há dois objetos fazendo uso da classe Carro. Os dois objetos estáticos (linha 28).

E a alocação no heap? É declarado um vetor dinâmico de objetos carros (arquivo *main.cpp*, linha 14). São dez ponteiros que inicialmente não fizeram nada ainda. Na linha 30, o primeiro ponteiro é alocado, invocando-se o método new e usando-se o construtor em que os parâmetros modelo, ano, preço e cor são passados. Na linha 31, o segundo ponteiro é alocado.

O método da classe mostrarTotalCarros é invocado, informando-nos que há quatro objetos fazendo uso da classe Carro. Os dois objetos estáticos, dois objetos dinâmicos (linha 36).

Podemos destruir esses dois novos objetos. Observe as linhas 38 e 39 do arquivo *main.cpp*. Pronto! Os ponteiros estão liberados caso queiram apontar para outros objetos. Na linha 43, o método da classe nos alerta: somente os dois objetos estão fazendo uso de carro. Três ponteiros alocam três outros carros (linha 42 e 46). Após alocar no heap, não podemos nos esquecer de desalocar os ponteiros (linhas 53-55).

Codificando um carro em Python

Vamos aos códigos em Python. A seguir, o arquivo *carro.py*.

```
01  #!/usr/bin/python3
02  # coding: utf-8
03  """Classe Carro """
04
05
06  class Carro(object):
07      """Cadastro de Carro"""
08      totalCarros = 0
09
10      def __init__(self, modelo=None, ano=None, preco=None, cor=No
11          """ Construtor """
12          if modelo is None:
13              self.modelo = "Fusca"
14              self.vel_max = 80
15              self.ano = 1981
16              self.preco = 5000
17              self.cor = "Azul"
18          else:
19              self.modelo = modelo
20              self.vel_max = 180
21              self.ano = ano
22              self.preco = preco
23              self.cor = cor
24          self.__class__.totalCarros += 1
25
26      def __del__(self):
27          """ Destrutor """
28          self.__class__.totalCarros -= 1
29          print ("Removendo {}: endereco {}".format(self.modelo, i
30
31      def get_modelo(self):
32          """ Retorna o modelo do automóvel """
33          return str(self.modelo)
34
35      def set_modelo(self, modelo):
36          """ Atribui um novo modelo ao automóvel """
```

```
37          if type(modelo) is str:
38              self.modelo = modelo
39
40      def get_cor(self):
41          """ Retorna a cor do automóvel """
42          return str(self.cor)
43
44      def set_cor(self, nova_cor):
45          """ Atribui uma nova cor ao automóvel """
46          if type(nova_cor) is str:
47              self.cor = nova_cor
48
49      def get_ano(self):
50          """ Retorna o ano do automóvel """
51          return str(self.ano)
52
53      def set_ano(self, novo_ano):
54          """ Atribui uma nova cor ao automóvel """
55          if type(novo_ano) is int:
56              self.ano = novo_ano
57          elif type(novo_ano) is str:
58              try:
59                  self.ano = int(novo_ano)
60              except ValueError:
61                  pass
62
63      def __repr__(self):
64          return '<{}: {} - {}>\n'.format(self.modelo, self.ano, s
65
66      @classmethod
67      def total(cls):
68          """ Método da classe. """
69          print ('Total de Carros: ' + str(cls.totalCarros))
70
71      @staticmethod
72      def stotal(classe_carro):
73          """ Método estático. """
74          print ('Total de Carros: ' + str(classe_carro.__class__.
```

A seguir, *main.py*.

```
01  #!/usr/bin/python3
02  # coding: utf-8
03  """ Exemplo do uso da classe Carro """
04
05  from carro import Carro
06
07  ESTACIONAMENTO = list()
08
09  for i in range(0, 5):
10      OBJ = Carro()
```

```
11      ESTACIONAMENTO.append(OBJ)
12
13  ESTACIONAMENTO[0].set_modelo('Monza')
14  ESTACIONAMENTO[0].set_cor('azul')
15  ESTACIONAMENTO[0].set_ano(1992)
16  ESTACIONAMENTO[0].set_ano('1993')
17  ESTACIONAMENTO[0].set_ano('aaaaa')
18
19  Carro.total()
20  del ESTACIONAMENTO[1]
21
22  for CARRO in ESTACIONAMENTO:
23      print (CARRO.get_modelo() + ' - ' + CARRO.get_ano())
24
25  Carro.stotal(ESTACIONAMENTO[0])
26
```

Espera-se, ao rodar este script, a saída a seguir.

```
Total de Carros: 5
Removendo Fusca: endereco 139659459248592
Monza - 1993
Fusca - 1981
Fusca - 1981
Fusca - 1981
Total de Carros: 4
Removendo Fusca: endereco 139659459249880
Removendo Fusca: endereco 139659459248704
Removendo Monza: endereco 139659459248536
Removendo Fusca: endereco 139659459249936
```

A sobrecarga de métodos ocorre quando há dois métodos com o mesmo nome, mas com assinaturas diferentes. No entanto faz sentido pensar sobrecarga em Python, sendo esta uma linguagem dinâmica? Vamos ao exemplo anterior: o atributo inteiro é setado com o método set_ano. O tipo passado como parâmetro pode ser qualquer um: uma string, um inteiro, um dicionário, uma lista ou outro objeto. Não é necessário ter sobrecarga. Porém, se eu espero um inteiro como parâmetro, como lidar com a liberdade que as linguagens dinâmicas me proporcionam? Tratando o parâmetro!

No arquivo *carro.py*, linha 53, o método set_ano recebe dois parâmetros: o primeiro parâmetro (self) é uma instância de classe. Em C++ (e em outras linguagens como Java, por exemplo) este parâmetro é passado implicitamente. Você o viu em C++ com o nome this. O segundo parâmetro é novo_ano. Qual o tipo do parâmetro novo_ano? Não sabemos. Assim, dentro do método, trabalhamos nele. Se o parâmetro for do tipo inteiro (arquivo *carro.py*, linhas 57 e 58), o atributo ano recebe o parâmetro novo_ano. Se o parâmetro for do tipo string (arquivo *carro.py*, linhas 59 a 63), tenta-se converter o parâmetro string em inteiro.

No código *main.py*, houve a tentativa de fazer errado o set_ano (linha 19), mas nenhuma alteração foi feita. A string ‘aaaa’ não é um inteiro. O método soube tratar este fato e nenhum erro ocorreu (arquivo *carro.py*, linhas 53-61).

Python também contém um construtor: o método __init__. Mas não há necessidade de sobrecarregar o construtor, como fizemos em C++. Em nossa implementação (arquivo *carro.py*, linhas 10 a 24), o construtor recebe self e os demais parâmetros setados como nulo (None). Ao identificar o atributo modelo como nulo, instancia-se um fusca azul. Caso contrário, recebem-se os parâmetros.

O método destrutor é o método __del__. Aqui vale um comentário especial. Em C++, caso você instancie dinamicamente um objeto e se esqueça de chamar o destrutor, um “lixo” fica na memória. Python tem um coletor de lixo (gargage collector). Assim, caso você não invoque o destrutor em Python, o interpretador faz

a faxina por você, desvinculando corretamente as memórias alocadas pelos objetos criados por você, embora possamos invocar o destrutor (arquivo *main.py*, linha 20).

Codificar o método da classe é possível com a palavra-chave @classmethod (arquivo *carro.py*, linhas 66 a 69).Observe a invocação desse método (arquivo *main.py*, linha 19). Ao invocar o método, nenhum parâmetro é passado, entretanto, na implementação, temos o parâmetro cls. Este parâmetro é um objeto que tem a própria classe. O atributo da classe (totalCarros) é incrementado a cada novo objeto instanciado no construtor e decrementado a cada objeto destruído, permitindo a esta variável contabilizar quantos carros temos.

Ao contrário dos métodos da classe, que têm a própria classe como parâmetro, os métodos estáticos (arquivo *carro.py*, linhas 71-74) não têm nenhum acesso à classe, aos objetos e aos atributos. Para conseguir realizar a tarefa do método da classe, foi necessário passar um objeto do tipo Carro como parâmetro (arquivo *main.py*, linha 25).

Síntese

Neste capítulo conhecemos alguns tópicos relevantes de orientação a objetos: sobrecarga, situação em que dois métodos têm o mesmo nome, porém assinaturas distintas; construtores e destrutores, métodos-chave ao se instanciar e desvincular os objetos; atributo de classe e método de classe, outro conceito importante que pode lhe poupar grandes esforços no futuro; alocação na pilha e no heap de objetos em C++. Por fim, reimplementamos o código do Carro em Python.

CAPÍTULO 6

Associações entre classes

"Nada é difícil se for dividido em pequenas partes."

HENRY FORD (1863-1947), EMPREENDEDOR AMERICANO

O que iremos aprender?

- Associação entre classes.
- Agregação e composição.

Que bom que você chegou até aqui. Agora que você conhece bem classe, objeto, método, atributo, vamos em frente. Não conhece? Pulou algumas páginas? Volte aos capítulos anteriores! Quando um programador me diz "Fiz meu código orientado a objetos", eu devolvo uma pergunta retórica: "Seu código está REALMENTE orientado a objetos?". Se a resposta for "Mais ou menos", eu já sei o que vou encontrar: um código estruturado dentro de uma única classe e um único objeto instanciado. Qual o grande problema nisso? Não permitir a reutilização eficiente de código e tornar a manutenção difícil conforme o software se expanda.

Um código orientado a objetos é um código que contém classes/objetos que se relacionam. Este relacionamento que ocorre entre as classes dos nossos códigos precisam ser estruturados com uso de associações.

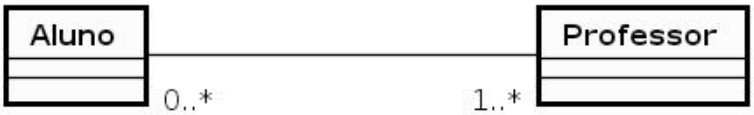


Figura 6.1 – Exemplo de associação.

A figura 6.1 apresenta um exemplo de associação. Nesse exemplo, a classe Aluno está associada à classe Professor. Além da linha que associa as duas classes, outra informação é passada: a multiplicidade desta associação. Multiplicidade tem relação com quantos objetos podem ser conectados. No exemplo, zero ou muitos alunos podem estar associados a um ou muitos professores.

Em alguns casos, é interessante criar um relacionamento "todo/parte" entre as classes. Neste tipo de associação, um objeto maior (todo) é formado ou outros objetos (parte) são formados. A essa associação especial damos o nome de agregação. Representa um relacionamento do tipo "tem", como na página 104.

A figura 6.2 apresenta um exemplo de agregação. No diagrama UML, o que diferencia uma associação de uma agregação é o losango do objeto-todo. Neste tipo especial de associação, Pessoa possui zero, um ou muitos apelidos. Um apelido sempre é atribuído a uma ou muitas pessoas. Neste caso, a Pessoa existe sem um apelido. Entretanto, Apelido (ou Apelidos) complementa as características de uma Pessoa. Há pessoas que não têm apelido algum. Há pessoas que têm um apelido e há pessoas (como eu, por exemplo) com uma vasta coleção de apelidos.

Figura 6.2 – Exemplo de agregação.

Uma variação da agregação é a associação do tipo composição. Neste caso de associação, o objeto todo é responsável pela criação e destruição de suas partes. A existência do objeto-parte não tem sentido sem o

objeto-todo.

O exemplo de composição da figura 6.3 nos mostra que uma sala de aula é composta de um ou mais alunos e um professor. Se o objeto sala de aula deixa de existir, o professor e os alunos deixam de existir também.

Você pode confundir a associação por agregação com associação por composição. Quando usar uma e quando usar outra? Depende das regras de negócio. Minha dica: o objeto-todo é prejudicado sem o objeto-parte? Se a resposta for sim, use composição. Se a resposta for não, agregação. Exemplo, imagine três objetos: carro, roda e som automotivo. Um relacionamento “todo-parte” existe entre estes objetos. Um carro (objeto-todo) tem quatro rodas, um som automotivo (além de várias outras partes). Sem rodas, o carro deixa de ser carro? Se sim, a classe carro é composta de rodas. Se não, a classe rodas é agregada à classe carro. O mesmo com o som automotivo. Para algumas pessoas, carro não é carro sem som automotivo.

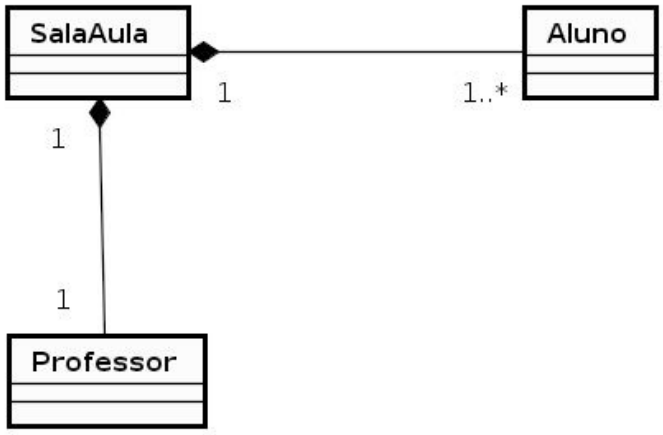


Figura 6.3 – Exemplo de composição.

Hackeando a caixa acoplada do vaso sanitário em C++

Vamos aos códigos para entendermos o conceito apresentado. Vamos codificar o objeto da figura 6.4. Sim, um vaso sanitário e uma caixa acoplada. Nosso código dará ênfase à caixa acoplada, esta maravilha da engenharia hidráulica. É tão simples que uma criança sabe operar: tem um único botão. Acabou o serviço, pressione-o! É tão simples que você poderia pensar em codificar com uma única classe e um único objeto.

Figura 6.4 – Vaso sanitário e caixa acoplada.

Não tão simples. A figura 6.5 ilustra uma caixa acoplada por dentro. Podemos observar que ela é composta de:

- Uma alavanca de acionamento – O botão pressionado no fim do serviço.
- Uma comporta de vedação – Uma tampa de borracha cuja função é abrir para a água descer e fechar para encher a caixa de água.
- Uma válvula de alimentação – responsável por encher de água.
- Uma boia – cuja função é fechar a válvula de alimentação assim que o nível da água estiver completo.

Ao pressionarmos a alavanca de acionamento, uma corda abre a comporta de vedação, permitindo que toda a água da caixa saia. Ao retornar à posição inicial, a comporta de vedação se fecha. Neste momento, a caixa está vazia. Entra em ação a válvula de alimentação. Cabe à válvula encher de água até a boia chegar ao nível completo novamente. Observe como cada objeto tem uma função específica; e todos compõem a caixa acoplada.

Figura 6.5 – Caixa acoplada por dentro.

No código a seguir, podemos ver o cabeçalho da alavanca de acionamento. Nenhuma outra funcionalidade é necessária além de acionar.

```
01  /*
02  Programa:   CaixaAcoplada
03  Arquivo:    alavancaAcionamento.h
04  */
05  #ifndef ALAVANCAACIONAMENTO_H
06  #define ALAVANCAACIONAMENTO_H
07
08  using namespace std;
09
10  class alavancaAcionamento {
11  private:
12
13  public:
14      alavancaAcionamento();
15      ~alavancaAcionamento();
16
17      void acionar();
18  };
19
20  #endif
```

No código a seguir, podemos ver o código da alavanca de acionamento.

```
01  /*
02  Programa:   CaixaAcoplada
03  Arquivo:    alavancaAcionamento.cpp
04  */
```

```
05  #include <stdio.h>
06  #include <stdlib.h>
07  #include <iostream>
08  #include "../includes/alavancaAcionamento.h"
09
10  using namespace std;
11
12  alavancaAcionamento::alavancaAcionamento() {
13      cout << "Construtor da alavanca de acionamento. " << endl;
14  }
15
16  alavancaAcionamento::~alavancaAcionamento() {
17      cout << "Destrutor da alavanca de acionamento. " << endl;
18  }
19
20  void alavancaAcionamento::acionar() {
21      cout << "Alavanca de acionamento ativada. " << endl;
22  }
```

A comporta de vedação tem as funções abrir e fechar, conforme codificado a seguir:

```
01  /*
02  Programa:   CaixaAcoplada
03  Arquivo:    comportaVedacao.h
04  */
05  #ifndef COMPORTAVEDACAO_H
06  #define COMPORTAVEDACAO_H
07
08  using namespace std;
09
10  class comportaVedacao {
11  private:
12
13  public:
14      comportaVedacao();
15      ~comportaVedacao();
16
17      float abrir();
18      void fechar();
19  };
20
21  #endif
```

A seguir, podemos ver o código da comporta de vedação.

```
01  /*
02  Programa:   CaixaAcoplada
03  Arquivo:    comportaVedacao.cpp
04  */
05  #include <stdio.h>
06  #include <stdlib.h>
07  #include <iostream>
08  #include "../includes/comportaVedacao.h"
```

```
09
10  using namespace std;
11
12  comportaVedacao::comportaVedacao() {
13      cout << "Construtor da comporta de Vedação. " << endl;
14  }
15
16  comportaVedacao::~comportaVedacao() {
17      cout << "Destrutor da comporta de Vedação. " << endl;
18  }
19
20  float comportaVedacao::abrir() {
21      cout << "Comporta de vedação aberta. Água saindo !!! " << en
22      return 0;
23  }
24
25  void comportaVedacao::fechar() {
26      cout << "Comporta de vedação fechada." << endl;
27  }
```

A válvula de alimentação tem uma única função: deixar a água entrar na caixa na vazão máxima. Eu não sei qual a vazão de uma válvula dessa, mas vamos imaginar que seja de 0,4 litros por segundo.

```
01  /*
02  Programa:   CaixaAcoplada
03  Arquivo:    valvulaAlimentacao.h
04  */
05  #ifndef VALVULAALIMENTACAO_H
06  #define VALVULAALIMENTACAO_H
07
08  using namespace std;
09
10  class valvulaAlimentacao {
11  private:
12      float capacidadeVazao = 0.4;  // litros por segundo
13  public:
14      valvulaAlimentacao();
15      ~valvulaAlimentacao();
16
17      float getCapacidadeVazao();
18  };
19
20  #endif
```

A seguir, podemos ver o código da válvula de alimentação.

```
01  /*
02  Programa:   CaixaAcoplada
03  Arquivo:    valvulaAlimentacao.cpp
04  */
05  #include <stdio.h>
06  #include <stdlib.h>
07  #include <iostream>
```

```
08  #include "../includes/valvulaAlimentacao.h"
09
10  using namespace std;
11
12  valvulaAlimentacao::valvulaAlimentacao() {
13      cout << "Construtor da válvula de alimentação. " << endl;
14  }
15
16  valvulaAlimentacao::~valvulaAlimentacao() {
17      cout << "Destrutor da válvula de alimentação. " << endl;
18  }
19
20  float valvulaAlimentacao::getCapacidadeVazao() {
21      return this->capacidadeVazao;
22  }
```

A caixa acoplada é o objeto-todo deste sistema. A seguir, o cabeçalho da caixa acoplada.

```
01  /*
02  Programa:   CaixaAcoplada
03  Arquivo:    CaixaAcoplada.h
04  */
05  #ifndef CAIXAACOPLADA_H
06  #define CAIXAACOPLADA_H
07
08  #include "alavancaAcionamento.h"
09  #include "comportaVedacao.h"
10  #include "valvulaAlimentacao.h"
11
12  using namespace std;
13
14  class CaixaAcoplada {
15  private:
16      float nivelAgua;
17      const float nivelMaximo = 6.0;  // litros
18
19      alavancaAcionamento* alavanca;
20      comportaVedacao* comporta;
21      valvulaAlimentacao* entradaAgua;
22
23  public:
24      CaixaAcoplada();
25      ~CaixaAcoplada();
26
27      void encherCaixa();
28      void acionar();
29  };
30
31  #endif
```

A seguir, o cabeçalho da caixa acoplada.

```
01  /*
```

```
02  Programa:   CaixaAcoplada
03  Arquivo:    CaixaAcoplada.cpp
04  */
05  #include <stdio.h>
06  #include <stdlib.h>
07  #include <iostream>
08  #include "../includes/CaixaAcoplada.h"
09
10  using namespace std;
11
12  CaixaAcoplada::CaixaAcoplada() {
13
14      alavanca = new alavancaAcionamento();
15      comporta = new comportaVedacao();
16      entradaAgua = new valvulaAlimentacao();
17
18      this->nivelAgua = 0;
19      this->encherCaixa();
20  }
21
22  CaixaAcoplada::~CaixaAcoplada() {
23      this->nivelAgua = 0;
24
25      delete alavanca;
26      delete comporta;
27      delete entradaAgua;
28  }
29
30  void CaixaAcoplada::encherCaixa() {
31      while (this->nivelAgua <= this->nivelMaximo) {
32                      cout  <<  "Nível  da  água  em:  "  <<  this-
>nivelAgua << endl;
33                     this->nivelAgua  =  nivelAgua  +  entradaAgua-
>getCapacidadeVazao();
34      }
35  }
36
37  void CaixaAcoplada::acionar() {
38      alavanca->acionar();
39      this->nivelAgua = comporta->abrir();
40      cout << "Nível da água em: " << this->nivelAgua << endl;
41      cout << "****************************************" << endl;
42      comporta->fechar();
43      this->encherCaixa();
44  }
```

A seguir, o arquivo *main.cpp*:

```
01  /*
02      Programa: CaixaAcoplada
03      Arquivo:  Main.cpp
```

```
04  */
05  #include <iostream>
06  #include <string>
07  #include "includes/CaixaAcoplada.h"
08
09  using namespace std;
10
11  int main() {
12      CaixaAcoplada *vaso = new CaixaAcoplada();
13      vaso->acionar();
14      delete vaso;
15
16      return 0;
17  }
```

Para melhor ilustrar como é o funcionamento da nossa caixa acoplada, faremos uso de um diagrama de sequência, que é um diagrama comportamental da UML cuja função é mostrar a ordem temporal das mensagens trocadas entre os objetos.

Na figura 6.6, ao instanciarmos o objeto da caixa acoplada, os objetos referentes à alavanca de acionamento, comporta de vedação e válvula de alimentação são instanciados também. Na etapa 4, o método encherCaixa() é invocado; uma mensagem é enviada ao objeto da válvula de alimentação solicitando a capacidade de vazão. Na etapa 5, a descarga é acionada. Uma mensagem é enviada para o objeto da alavanca de acionamento. Uma mensagem é enviada à comporta de vedação, permitindo à água descer pela descarga e repondo o nível da água. Na sequência, uma mensagem é enviada à comporta de vedação para o fechamento e novamente encher a caixa de água.

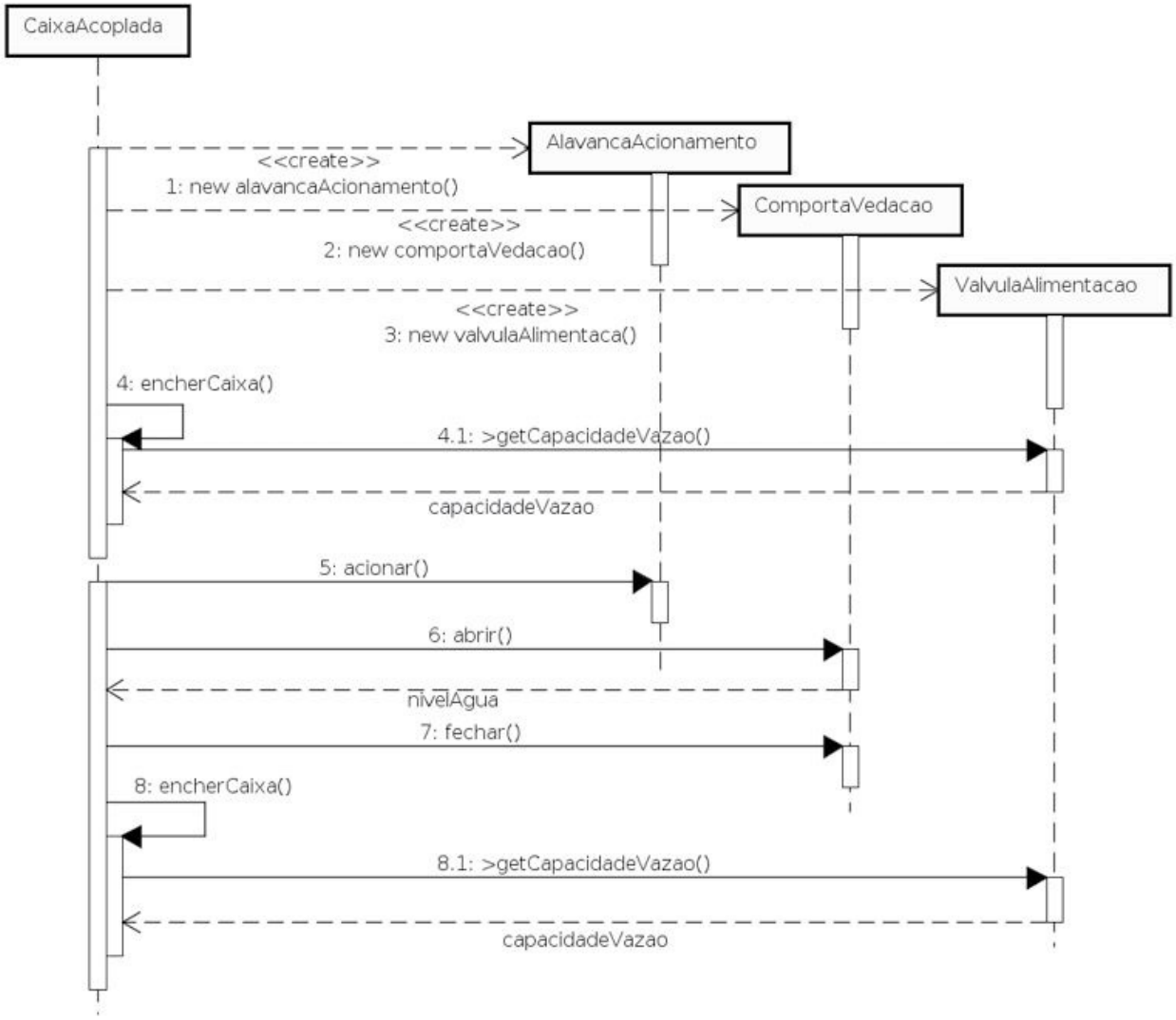


Figura 6.6 – Diagrama de sequência da caixa acoplada.

A execução do programa é apresentada a seguir:

```
Construtor da alavanca de acionamento.
Construtor da comporta de vedação.
Construtor da válvula de alimentação.
Nível da água em: 0
Nível da água em: 0.4
Nível da água em: 0.8
Nível da água em: 1.2
Nível da água em: 1.6
Nível da água em: 2
Nível da água em: 2.4
Nível da água em: 2.8
Nível da água em: 3.2
Nível da água em: 3.6
Nível da água em: 4
Nível da água em: 4.4
Nível da água em: 4.8
```

```
Nível da água em: 5.2
Nível da água em: 5.6
Alavanca de acionamento ativada.
Comporta de vedação aberta. Água saindo !!!
Nível da água em: 0
**************************************
Comporta de vedação fechada.
Nível da água em: 0
Nível da água em: 0.4
Nível da água em: 0.8
Nível da água em: 1.2
Nível da água em: 1.6
Nível da água em: 2
Nível da água em: 2.4
Nível da água em: 2.8
Nível da água em: 3.2
Nível da água em: 3.6
Nível da água em: 4
Nível da água em: 4.4
Nível da água em: 4.8
Nível da água em: 5.2
Nível da água em: 5.6
Destrutor da alavanca de acionamento.
Destrutor da comporta de vedação.
Destrutor da válvula de alimentação.
```

O código está pronto. Ficou claro como o uso de associações entre classes pode ser bom. E neste momento quero lhe propor um desafio de requisitos:

• Requisito 1 – Mudar a velocidade de enchimento da caixa. Precisamos que seja mais rápido.

• Requisito 2 – Mudar o comportamento da caixa. Precisamos de uma privada inteligente. Teremos opção 1 quando o usuário fizer a opção 1 e opção 2 para um serviço mais completo por parte do usuário.

• Requisito 3 – Contador de descargas. Como a água é um problema mundial, nossa caixa acopladora terá um contador de descargas.

Para o primeiro requisito, vamos pensar: se a velocidade para encher a caixa precisa ser maior, o que é necessário é permitir que mais água entre em um menor espaço de tempo. Alterando-se a linha 12 do arquivo *valvulaAlimentacao.h* para um número maior, aumenta-se a vazão da água ao encher a caixa.

```
12      float capacidadeVazao = 1.1;  // litros por segundo
```

Para o segundo requisito, quais objetos relacionam-se com esta funcionalidade? Os objetos caixa acoplada, nosso objeto-todo. A comporta de vedação, que terá como nova atribuição não deixar toda a água sair, mas apenas metade da caixa quando o usuário fizer a opção 1. A alavanca de acionamento terá como responsabilidade observar qual a opção escolhida do usuário e alertá-lo das opções disponíveis: 1 e 2.

A caixa acoplada precisa ter um método acionar esperando um parâmetro inteiro. A sobrecarga do método acionar pode ser observada no código a seguir:

```
// Método adicionado para atendimento do requisito 2
void CaixaAcoplada::acionar(int opcao) {
    alavanca->acionar(opcao);
    if (opcao == 1) {
        this->nivelAgua = comporta->abrir(this->nivelMaximo/2);
                  cout  <<  "Nível  da  água  em:  "  <<  this-
```

```
>nivelAgua << endl;
            cout << "****************************************" << en
            comporta->fechar();
            this->encherCaixa();
        }
        if (opcao == 2) {
            this->acionar();
        }
    }
```

Na comporta de vedação é necessário um método que receba um parâmetro informando a quantidade de água a ser liberada. A sobrecarga do método abrir pode ser observada a seguir:

```
    // Método adicionado para atendimento do requisito 2
    float comportaVedacao::abrir(float vazao) {
              cout << "Comporta de vedação aberta. Liberando "
<< vazao << " litros de água" << endl;
        return vazao;
    }
```

Para a alavanca de acionamento foi necessário sobrescrever o método acionar:

```
    void alavancaAcionamento::acionar(int opcao) {
        cout << "Privada Inteligente. " << endl;
        if ((opcao == 1) || (opcao == 2)) {
            cout << "Opção : " << opcao << " selecionada" << endl;
        } else {
            cout << "Opção desconhecida. Pressione 1 ou 2." << endl;
        }
    }
```

Para o terceiro requisito, alteraremos os objetos caixa acoplada (o objeto-todo) e a alavanca de acionamento. Para a caixa acoplada, o método getRelatorio() foi criado.

```
    // Método adicionado para atendimento do requisito 3
    void CaixaAcoplada::getRelatorio() {
        alavanca->getRelatorio();
    }
```

As alterações na alavanca de acionamento foram mais significativas para atender ao requisito 3. Foram implementados dois contadores do tipo inteiro na classe. Nos métodos acionar, a cada descarga acionada, os contadores são incrementados. Por fim, foi desenvolvido um novo método para o relatório.

```
    void alavancaAcionamento::getRelatorio() {
        cout << "\n\nRelatório de uso da Privada Inteligente. " << e
                        cout   <<   "Opção   1:   "   <<   this-
>contador_1 << " descargas aplicadas" << endl;
                        cout   <<   "Opção   2:   "   <<   this-
>contador_2 << " descargas aplicadas" << endl;
        cout << "\n\n ";
    }
```

O arquivo *main.cpp* foi alterado para verificar os requisitos que foram atendidos sem alterar as funcionalidades anteriores.

```
    CaixaAcoplada *vaso = new CaixaAcoplada();
    vaso->acionar();
    vaso->acionar(1);
    vaso->acionar(2);
```

```
vaso->acionar(3);
vaso->getRelatorio();
delete vaso;
```

A execução do programa com os requisitos 1, 2 e 3 atendidos pode ser observada a seguir.

```
Construtor da alavanca de acionamento.
Construtor da comporta de Vedação.
Construtor da válvula de alimentação.
Nível da água em: 0
Nível da água em: 1.1
Nível da água em: 2.2
Nível da água em: 3.3
Nível da água em: 4.4
Nível da água em: 5.5
Alavanca de acionamento ativada.
Comporta de vedação aberta. Água saindo !!!
Nível da água em: 0
****************************************
Comporta de vedação fechada.
Nível da água em: 0
Nível da água em: 1.1
Nível da água em: 2.2
Nível da água em: 3.3
Nível da água em: 4.4
Nível da água em: 5.5
Privada Inteligente.
Opção : 1 selecionada
Opção desconhecida. Pressione 1 ou 2.
Comporta de vedação aberta. Liberando 3 litros de água
Nível da água em: 3
****************************************
Comporta de vedação fechada.
Nível da água em: 3
Nível da água em: 4.1
Nível da água em: 5.2
Privada Inteligente.
Opção : 2 selecionada
Alavanca de acionamento ativada.
Comporta de vedação aberta. Água saindo !!!
Nível da água em: 0
****************************************
Comporta de vedação fechada.
Nível da água em: 0
Nível da água em: 1.1
Nível da água em: 2.2
Nível da água em: 3.3
Nível da água em: 4.4
Nível da água em: 5.5
Privada Inteligente.
Opção desconhecida. Pressione 1 ou 2.
```

```
Relatório de uso da Privada Inteligente.
Opção 1: 1 descargas aplicadas
Opção 2: 2 descargas aplicadas
 Destrutor da alavanca de acionamento.
Destrutor da comporta de Vedação.
Destrutor da válvula de alimentação.
```

Hackeando a caixa acoplada do vaso sanitário em Python

Com todos os requisitos, vamos ao código Python! O diagrama de classe na figura 6.7 nos apresenta uma visão do que é a classe caixa acoplada e quais são os objetos-parte.

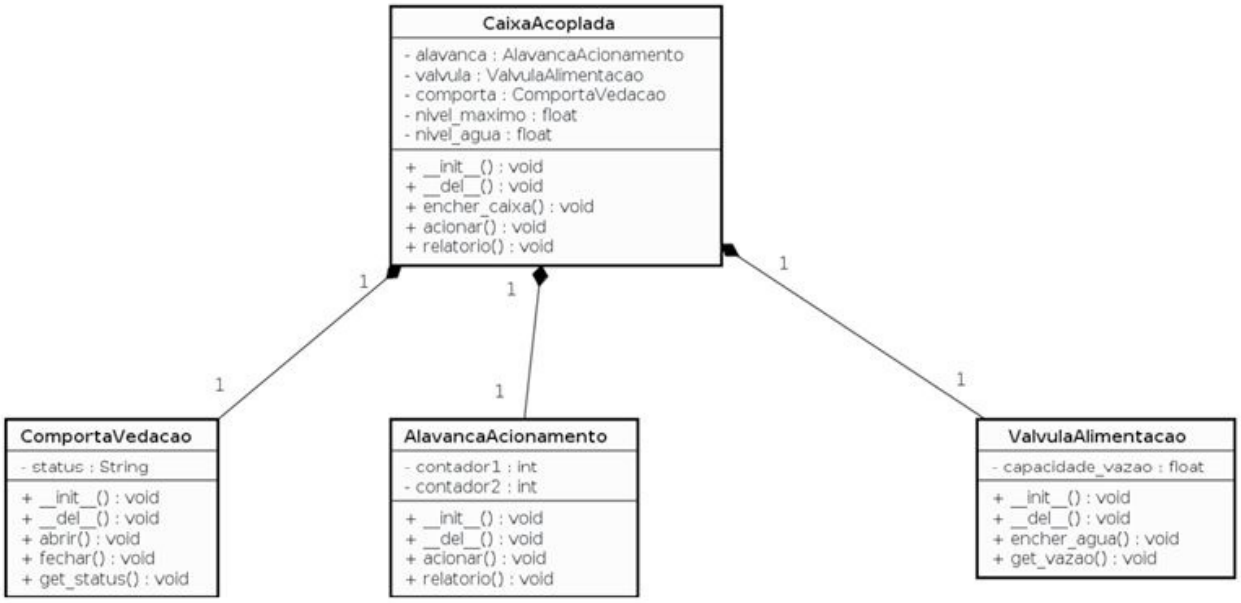


Figura 6.7 – Diagrama de classes da caixa acoplada.

A seguir, o código da caixa acoplada. Observe como os objetos são instanciados no construtor (método __init__).

```
01  #!/usr/bin/python3
02  # coding: utf-8
03  """Classe CaixaAcoplada """
04
05  from AlavancaAcionamento import AlavancaAcionamento
06  from ValvulaAlimentacao import ValvulaAlimentacao
07  from ComportaVedacao import ComportaVedacao
08
09
10  class CaixaAcoplada(object):
11      """ Caixa acoplada do vaso sanitário """
12
13      def __init__(self):
14          """ Construtor instancia outros objetos, seta o nível ""
15          print("Construtor da caixa do vaso sanitário")
16          self.alavanca = AlavancaAcionamento()
17          self.valvula = ValvulaAlimentacao()
18          self.comporta = ComportaVedacao()
```

```
19          self.nivel_maximo = 6.0
20          self.nivel_agua = 0
21          self.encher_caixa()
22
23      def __del__(self):
24          """ Destrutor """
25          print("Removendo caixa acoplada: endereco {}".format(id(
26
27      def encher_caixa(self):
28          """ Encher a caixa de água """
29          print("Encher a caixa d'água.")
30          while self.nivel_agua < self.nivel_maximo:
31              self.nivel_agua = self.nivel_agua + self.valvula.get
32              if self.nivel_agua > self.nivel_maximo:
33                  self.nivel_agua = self.nivel_maximo
34              print("Nível de água: " + str(round(self.nivel_agua,
35
36      def acionar(self, opcao=None):
37          """ Acionar a descarga """
38          print("Acionado o vaso sanitário.")
39
40          if type(opcao) == int:
41              if opcao == 1:
42                  print("Número 1. Gastar pouca água")
43                  self.alavanca.acionar(opcao)
44                  self.comporta.abrir()
45                  self.nivel_agua = self.nivel_maximo / 2
46                  self.comporta.fechar()
47                  self.valvula.encher_agua()
48                  self.encher_caixa()
49              elif opcao == 2:
50                  print("Número 2. ")
51                  self.alavanca.acionar(opcao)
52                  self.comporta.abrir()
53                  self.nivel_agua = 0
54                  self.comporta.fechar()
55                  self.valvula.encher_agua()
56                  self.encher_caixa()
57              else:
58                  print("Opção desconhecida.")
59          else:
60              print("Não é o vaso inteligente..")
61              self.alavanca.acionar()
62              self.comporta.abrir()
63              self.nivel_agua = 0
64              self.comporta.fechar()
65              self.valvula.encher_agua()
66              self.encher_caixa()
67
68      def relatorio(self):
```

```
69          """ Método para emitir relatório de uso """
70          self.alavanca.relatorio()
71
```

O código da alavanca de acionamento é este:

```
01  #!/usr/bin/python3
02  # coding: utf-8
03  """ Classe alavanca de acionamento. Aquele botão da caixa d'água
04
05
06  class AlavancaAcionamento(object):
07      """ alavanca de acionamento """
08
09      def __init__(self):
10          """ Construtor """
11          print("Construtor da alavanca de acionamento")
12          self.contador1 = 0
13          self.contador2 = 0
14
15      def __del__(self):
16          """ Destrutor """
17          print("Removendo alavanca: endereco {}".format(id(self))
18
19      def acionar(self, opcao=None):
20          """ Acionado a alavanca de acionamento. Pressionaram o b
21          print("alavanca de acionamento ativada. ")
22          if type(opcao) == int:
23              if opcao == 1:
24                  self.contador1 += 1
25              else:
26                  self.contador2 += 1
27          else:
28              self.contador2 += 1
29
30      def relatorio(self):
31          """ Relatório de uso de água """
32          print("Relatório de uso de água. ")
33          print("")
34          print("Número de descargas Opção 1: " + str(self.contado
35          print("Número de descargas Opção 2: " + str(self.contado
36
```

A classe da comporta de vedação é mostrada a seguir.

```
01  #!/usr/bin/python3
02  # coding: utf-8
03  """ Classe comporta de vedação """
04
05
06  class ComportaVedacao(object):
07      """comporta de vedação"""
08
```

```
09      def __init__(self):
10          """ Construtor """
11          print("Construtor da comporta de vedação")
12          self.status = "FECHADO"
13
14      def __del__(self):
15          """ Destrutor """
16          print("Removendo comporta de vedação: endereco {}".forma
17
18      def abrir(self, vazao=None):
19          """ Abertura da comporta de vedação """
20          if vazao == None:
21              print("Comporta de vedação aberta. Saindo toda de ág
22          else:
23              print("Comporta aberta. Saindo "+ str(vazao) + "de á
24
25          self.status = "ABERTO"
26
27      def fechar(self):
28          """ Fechamento da comporta de vedação. Água Saindo... ""
29          print("Comporta de vedação fechada.")
30          self.status = "FECHADO"
31
32      def get_status(self):
33          """ Status da comporta de vedação (Aberta ou fechada) ""
34          return self.status
35
```

A classe referente à válvula de alimentação é esta:

```
01  #!/usr/bin/python3
02  # coding: utf-8
03  """ Válvula de alimentação """
04
05
06  class ValvulaAlimentacao():
07      """ Válvula de alimentação """
08
09      def __init__(self):
10          """ Construtor """
11          print("Construtor da válvula de alimentação")
12          self.capacidade_vazao = 1.1
13
14      def __del__(self):
15          """ Destrutor """
16          print("Removendo válvula de alimentação: endereco {}".fo
17
18      def encher_agua(self):
19          """ Descarga inteligente. Opção 1: pipi Opção 2: popô ""
20          print("Vazão :" + str(self.capacidade_vazao) + " litros/
21
```

```
22      def get_vazao(self):
23          """ Retorna a capacidade de vazão do vaso """
24          return self.capacidade_vazao
25
```

E, por fim, o código *main.py*, que faz uso da caixa acoplada e dos métodos disponíveis.

```
01  #!/usr/bin/python3
02  # coding: utf-8
03  """ Exemplo do uso da caixa acoplada """
04
05  from CaixaAcoplada import CaixaAcoplada
06
07  VASO = CaixaAcoplada()
08  VASO.acionar()
09  print('\n\n')
10  VASO.acionar(1)
11  print('\n\n')
12  VASO.acionar(2)
13  print('\n\n')
14  VASO.acionar(3) # descarga inválida
15  VASO.relatorio()
```

A execução do programa é exibida a seguir. O construtor da caixa acoplada instancia os objetos-parte. O destrutor do objeto-todo não é invocado. O destrutor dos objetos-parte também não são invocados. Mas, ao final da execução, os destrutores são invocados pelo interpretador Python. Obrigado, interpretador. Sem o gargage collection, uma parte da memória ficaria abandonada após a execução do programa.

```
    Construtor da caixa do vaso sanitário
    Construtor da alavanca de acionamento
    Construtor da válvula de alimentação
    Construtor da comporta de vedação
    Encher a caixa d'água.
    Nível de água: 1.1
    Nível de água: 2.2
    Nível de água: 3.3
    Nível de água: 4.4
    Nível de água: 5.5
    Nível de água: 6.0
    Acionado o vaso sanitário.
    Não é o vaso inteligente..
    alavanca de acionamento ativada.
    Comporta de vedação aberta. Saindo toda de água.
    Comporta de vedação fechada.
    Vazão :1.1 litros/seg
    Encher a caixa d'água.
    Nível de água: 1.1
    Nível de água: 2.2
    Nível de água: 3.3
    Nível de água: 4.4
    Nível de água: 5.5
    Nível de água: 6.0
    Acionado o vaso sanitário.
```

```
Número 1. Gastar pouca água
alavanca de acionamento ativada.
Comporta de vedação aberta. Saindo toda de água.
Comporta de vedação fechada.
Vazão :1.1 litros/seg
Encher a caixa d'água.
Nível de água: 4.1
Nível de água: 5.2
Nível de água: 6.0
Acionado o vaso sanitário.
Número 2.
alavanca de acionamento ativada.
Comporta de vedação aberta. Saindo toda de água.
Comporta de vedação fechada.
Vazão :1.1 litros/seg
Encher a caixa d'água.
Nível de água: 1.1
Nível de água: 2.2
Nível de água: 3.3
Nível de água: 4.4
Nível de água: 5.5
Nível de água: 6.0
Acionado o vaso sanitário.
Opção desconhecida.
Relatório de uso de água.
Número de descargas Opção 1: 1
Número de descargas Opção 2: 2
Removendo caixa Acoplada: endereco 140696991465600
Removendo alavanca: endereco 140696991465656
Removendo válvula de alimentação: endereco 140696991469240
Removendo comporta de vedação: endereco 140696991594256
```

Síntese

Neste capítulo aprendemos como os objetos se relacionam. O relacionamento entre os objetos em uma aplicação pode ser por associação ou por herança (nosso próximo assunto). Conhecemos dois tipos especiais de associação: agregação e composição. Codificamos um objeto real para observar a composição de um objeto-todo por outros objetos-parte. Ficou claro como a modularidade e a divisão de responsabilidades entre as classes podem facilitar o atendimento de novos requisitos. Aos olhos de um programador externo, a classe referente ao objeto caixa acoplada é uma classe simples, com poucas funcionalidades.

CAPÍTULO 7

Herança

“Tudo que se acende apaga.
Tudo que está dentro sai.
Tudo que sobe desce.
E tudo se encaixa.”
Tudo
BANDA PATO FU

O que iremos aprender?

- Associação entre classes por herança.
- Sobreposição.
- Encapsulamento.

No capítulo anterior, vimos como ocorrem associações entre classes. Analisamos o relacionamento por agregação e por composição. Esta relação é conhecida por meio de um relacionamento “tem”.

- Uma pessoa tem um coração, tem dois pulmões, tem um fígado.
- Uma caixa acoplada tem uma alavanca de acionamento, uma comporta de vedação e uma válvula de alimentação.

A relação de herança se baseia no princípio de que toda a codificação mais genérica pode ser transmitida para as classes mais específicas. Esta relação é conhecida por meio do relacionamento “é-um”.

- Um cachorro é um mamífero. Um gato é um mamífero. Um coelho é um mamífero.
- Uma Ferrari é um carro. Uma BMW é um carro. Um fusca é um carro.

No exemplo, cachorro, gato e coelho são diferentes. Mas uma classe mamífero tem características comuns a todos esses animais. Não é necessário reimplementar alguns atributos e métodos já codificados na classe mamífero. Todos têm características comuns: mamam quando filhotes.

Carro tem características de todos os carros: rodas, cor, preço, ano de fabricação, dentre outros. Embora haja diferenças (muitas), Ferrari, BMW e fusca têm características comuns. A modelagem desse tipo de relacionamento segue o padrão apresentado na figura 7.1.

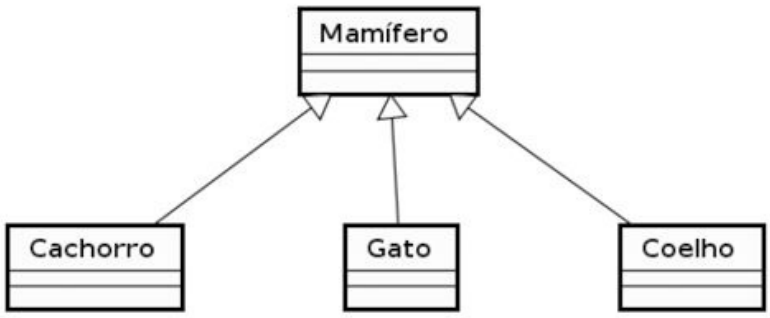


Figura 7.1 – Diagrama de classes UML demonstrando herança.

Na figura 7.1 observamos a classe Mamifero. Esta classe é conhecida como classe-base. Dependendo da bibliografia que você pesquisar, esta classe é conhecida como superclasse, classe-mãe, classe-pai. As classes Cachorro, Gato e Coelho são as classes derivadas. Também são conhecidas como classes-filhas ou subclasses. Os atributos e métodos implementados na classe mamífero podem ser herdados das demais classes.

Como um conjunto de classes pode fazer reúso de código com herança? Observe nessas classes quais atributos e funcionalidades são comuns. Para essas características e esses comportamentos comuns, reúna-os em uma classe mais genérica.

Sistema político brasileiro

Para um melhor entendimento de herança, vamos desenvolver o aplicativo Políticos. Em 2010, o deputado federal mais votado do Estado de São Paulo ganhou a eleição com o seguinte slogan de campanha: “O que é que faz um deputado federal? Na realidade, eu não sei. Mas vote em mim que eu te conto.” Não contou e foi reeleito em 2014. Agora iremos saber o que faz um deputado federal.

O modelo de governabilidade do Brasil é baseado na separação de poderes proposta por Montesquieu. Temos o poder legislativo, responsável por propor leis de interesse para a sociedade. Temos o poder executivo, responsável pela administração e aplicação dos recursos públicos, e o poder judiciário.

O Congresso Nacional, órgão do poder legislativo, é composto com o Senado e a Câmara dos Deputados. No Senado há 81 senadores, três para cada Estado, com mandato de oito anos. No Congresso há 513 deputados federais com mandato de quatro anos.

Toda proposta de lei é votada pela Câmara dos Deputados e pelo Senado. Após esta etapa, a lei é sancionada (ou não) pelo presidente, representante do poder executivo. Temos a figura do presidente e um vice-presidente.

Na esfera estadual, temos os deputados estaduais como representantes do poder legislativo estadual. No poder executivo estadual, temos as figuras do governador e do vice-governador. Na esfera municipal, temos os vereadores como representantes do poder legislativo e o prefeito e o vice-prefeito como representantes do poder executivo.

Agora que você já sabe o que faz um deputado e como funciona o modelo de governabilidade brasileiro, vote em mim nas próximas eleições.

Sistema político brasileiro em C++

O que senador, deputado federal, deputado estadual, vereador, presidente, governador e prefeito têm em comum? Todos são políticos. Quais os atributos comuns a todos os políticos? Nome, partido político, Estado, e cada um exerce uma função específica. Há outros atributos, mas vamos nos limitar a estes. Em C++, desenvolvemos a classe Politico e duas classes-filhas somente: prefeito e vereador. Poderíamos criar mais classes-filhas, uma para cada “categoria” de político, porém, como nosso foco é entender herança, ficaremos apenas com a esfera municipal (Figura 7.2).

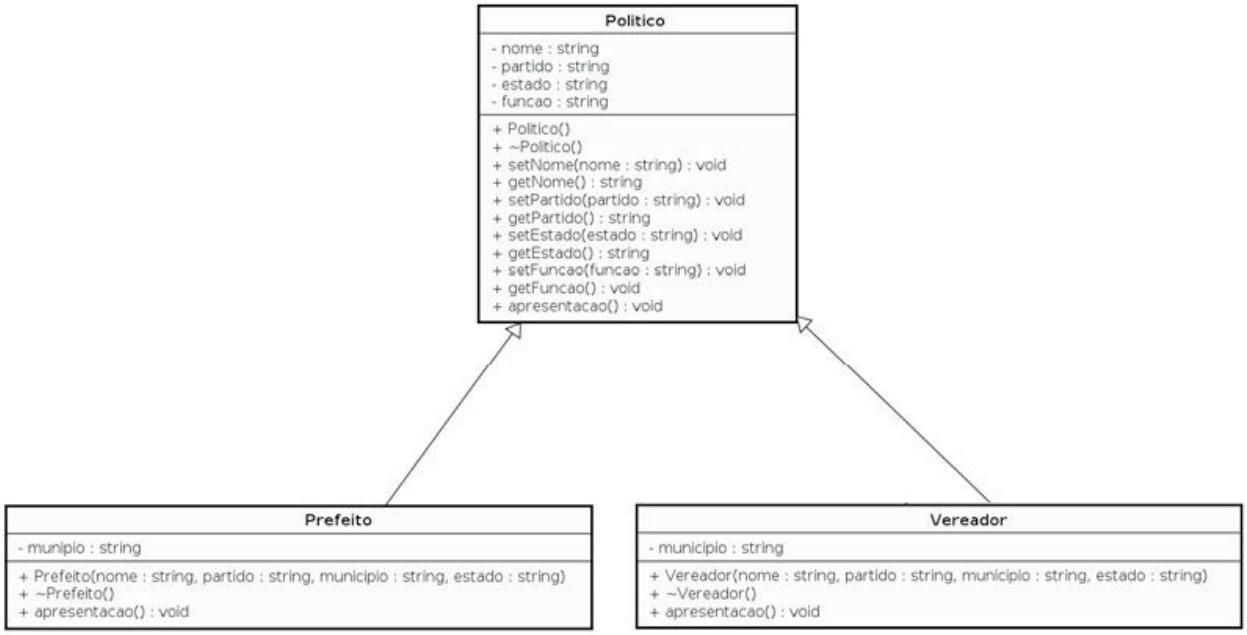


Figura 7.2 – Diagrama de classes dos políticos atuantes na esfera municipal.

A classe Politico é a classe-base para todas as outras. Toda a codificação da classe Politico será reaproveitada nas classes-filhas. A seguir, o cabeçalho da classe Politico.

```
01  /*
02  Programa:   Políticos
03  Arquivo:    Politico.h
04  */
05  #ifndef POLITICO_H
06  #define POLITICO_H
07
08  using namespace std;
09
10  class Politico {
11  private:
12      string nome;
13      string partido;
14      string estado;
15      string funcao;
16  public:
17      Politico();
18      ~Politico();
19
20      void setNome(string);
21      string getNome();
22      void setPartido(string);
23      string getPartido();
24      void setEstado(string);
25      string getEstado();
26      void setFuncao(string);
27      string getFuncao();
28      void apresentacao();
```

```
29  };
30
31  #endif
```

Definida no arquivo-cabeçalho a estrutura da classe Politico, no arquivo *Político.cpp*, codificamos os métodos definidos anteriormente.

```
01  /*
02  Programa:   Políticos
03  Arquivo:    Politico.cpp
04  */
05  #include <stdio.h>
06  #include <stdlib.h>
07  #include <iostream>
08  #include "../includes/Politico.h"
09
10  using namespace std;
11
12  Politico::Politico() {
13  }
14
15  Politico::~Politico() {
16  }
17
18  void Politico::setNome(string nome){
19      this->nome = nome;
20  }
21
22  string Politico::getNome(){
23      return this->nome;
24  }
25
26  void Politico::setPartido(string partido){
27      this->partido = partido;
28  }
29
30  string Politico::getPartido(){
31      return this->partido;
32  }
33
34  void Politico::setEstado(string estado){
35      this->estado = estado;
36  }
37
38  string Politico::getEstado(){
39      return this->estado;
40  }
41
42  void Politico::setFuncao(string funcao){
43      this->funcao = funcao;
44  }
```

```
45
46  string Politico::getFuncao(){
47      return this->funcao;
48  }
49
50  void Politico::apresentacao(){
51      cout << "Olá, sou "<< this->getNome() << endl;
52      cout << "Meu partido é "<< this->getPartido() << endl;
53  }
```

Na instância municipal, temos as figuras do prefeito e do vereador. A seguir, a definição da classe Prefeito.

```
01  /*
02  Programa:   Políticos
03  Arquivo:    Prefeito.h
04  */
05  #ifndef PREFEITO_H
06  #define PREFEITO_H
07  #include "Politico.h"
08
09  using namespace std;
10
11  class Prefeito : public Politico {
12  private:
13      string municipio;
14  public:
15      Prefeito(string,string,string,string);
16      ~Prefeito();
17
18      void setMunicipio(string);
19      string getMunicipio();
20      void apresentacao();
21  };
22
23  #endif
```

Observe na linha 11 do arquivo Prefeito.h o uso de herança. A seguir, o arquivo Prefeito.cpp.

```
01  /*
02  Programa:   Políticos
03  Arquivo:    Prefeito.cpp
04  */
05  #include <stdio.h>
06  #include <stdlib.h>
07  #include <iostream>
08  #include "../includes/Prefeito.h"
09  using namespace std;
10
11  Prefeito::Prefeito(string nome,string partido,string municipio,
12      setNome(nome);
13      setPartido(partido);
14      setMunicipio(municipio);
15      setEstado(estado);
```

```
16      setFuncao("administrar o IPTU visando o melhor para a cidade
17  }
18
19  Prefeito::~Prefeito() {
20      setNome("");
21      setPartido("");
22      setEstado("");
23  }
24
25  void Prefeito::setMunicipio(string municipio){
26      this->municipio = municipio;
27  }
28
29  string Prefeito::getMunicipio(){
30      return this->municipio;
31  }
32
33  void Prefeito::apresentacao(){
34      Politico::apresentacao();
35      cout << "sou prefeito em " << getMunicipio() << " / " << get
36      cout << "Minha função é " << this->getFuncao() << endl;
37      cout << "============================" << endl;
38  }
```

No poder legislativo, temos o vereador, codificado a seguir.

```
01  /*
02  Programa:   Políticos
03  Arquivo:    Vereador.h
04  */
05  #ifndef VEREADOR_H
06  #define VEREADOR_H
07  #include "Politico.h"
08
09  using namespace std;
10
11  class Vereador : public Politico {
12  private:
13      string municipio;
14  public:
15      Vereador(string,string,string,string);
16      ~Vereador();
17
18      void setMunicipio(string);
19      string getMunicipio();
20      void apresentacao();
21  };
22
23  #endif
```

Após a definição feita anteriormente no arquivo Vereador.h, a seguir, a codificação, presente no arquivo Vereador.cpp.

```
01  /*
02  Programa:   Políticos
03  Arquivo:    Vereador.cpp
04  */
05  #include <stdio.h>
06  #include <stdlib.h>
07  #include <iostream>
08  #include "../includes/Vereador.h"
09  using namespace std;
10
11  Vereador::Vereador(string nome,string partido,string municipio,
12      setNome(nome);
13      setPartido(partido);
14      setMunicipio(municipio);
15      setEstado(estado);
16      setFuncao("propor leis municipais em benefício da população.
17  }
18
19  Vereador::~Vereador() {
20      setNome("");
21      setPartido("");
22      setEstado("");
23  }
24
25  void Vereador::setMunicipio(string municipio){
26      this->municipio = municipio;
27  }
28
29  string Vereador::getMunicipio(){
30      return this->municipio;
31  }
32
33  void Vereador::apresentacao(){
34      Politico::apresentacao();
35      cout << "sou vereador em " << getMunicipio() << " / " << get
36      cout << "Minha função é " << this->getFuncao() << endl;
37      cout << "============================" << endl;
38  }
```

Codificadas a classe-base (Politico) e as duas classes-filhas (Prefeito e Vereador), a seguir, observamos o código principal (Main.cpp) fazendo uso das classes.

```
01  /*
02  Programa:   Políticos
03  Arquivo:    Main.cpp
04  */
05  #include <iostream>
06  #include <string>
07
08  #include "includes/Prefeito.h"
09  #include "includes/Vereador.h"
```

```
10
11  using namespace std;
12
13  int main() {
14
15      Prefeito *pref = NULL;
16      Vereador *vered = NULL;
17
18
19      pref = new Prefeito("Odorico Paraguaçu","Partido do Povo", "
20      vered = new Vereador("Doroteia","Partido do Povo","Sucupira"
21
22      pref->apresentacao();
23      vered->apresentacao();
24  //    cout << endl << vered->municipio << endl;
25      return 0;
26  }
```

A execução do código pode ser vista a seguir. Um show de bons políticos da ficção.

```
Olá, sou Odorico Paraguaçu
Meu partido é Partido do Povo
sou prefeito em Sucupira / BA
Minha função é administrar o IPTU visando o melhor para a cidade
============================
Olá, sou Doroteia
Meu partido é Partido do Povo
sou vereador em Sucupira / BA
Minha função é propor leis municipais em benefício da população.
============================
```

As duas classes herdaram as funcionalidades codificadas na classe Politico. Esta associação facilita centralizar códigos na classe-base. Quando as classes derivadas necessitam de algum atributo um pouco mais específico, o codificam em sua classe. No caso das classes vereador e prefeito, armazenamos o município na própria classe. Armazenar o município ao qual foi eleito em nosso caso não é um atributo comum a todos os políticos em nosso sistema.

Sobreposição

A sobreposição de método (*override*) é o recurso por meio do qual uma classe derivada reescreve o método da classe-base a fim de atender a alguma particularidade. Em nosso exemplo, a classe-base se utiliza do método apresentacao(). As classes derivadas reimplementaram esse método para personalizar a mensagem de apresentação (Vereador.cpp, linhas 33-38, e Prefeito.cpp, linhas 33-38).

Não confunda sobreposição de método (*override*) com sobrecarga de métodos (*overload*). A sobrecarga é ter em uma mesma classe dois ou mais métodos com o mesmo nome, mas com parâmetros diferentes.

Encapsulamento e os métodos de acesso

Conhecemos dois métodos de acesso aos métodos e atributos: privado (*private*) e público (*public*). Para relembrar, altere o arquivo *Main.cpp* conforme mostrado a seguir:

```
24      cout << endl << vered->municipio << endl;
```

Tente recompilar o programa. Não conseguirá. Um erro como o apresentado a seguir lhe espera.

```
In file included from Main.cpp:13:0:
```

```
includes/Vereador.h: In function 'int main()':
includes/Vereador.h:9:12: error: 'std::string Vereador::municipi
string municipio;
           ^
Main.cpp:24:28: error: within this context
     cout << endl << vered->municipio << endl;
```

O erro é justificável. Ao tentar acessar um atributo privado (municipio) fora da classe (Vereador), o compilador acertadamente identificou um erro. Altere o arquivo *Vereador.h*, o conteúdo na linha 12. Troque a palavra private por public conforme apresentado a seguir. Em seguida, recompile o programa. Nenhum erro ocorre.

```
11  class Vereador : public Politico {
12  private:
13      string municipio;
14  public:
15      Vereador(string,string,string,string);
16      ~Vereador();
```

Concluímos que o método de acesso privado permite acesso somente dentro da classe. E o método de acesso público permite acesso tanto internamente quanto fora da classe. Há um terceiro método de acesso para investigarmos: o acesso protegido (protected).

Altere a linha 12 do método construtor no arquivo *Vereador.cpp* conforme o código a seguir. Tente recompilar o programa.

```
11  Vereador::Vereador(string nome,string partido,string municip
12      this->nome = nome;    // setNome(nome);
13      setPartido(partido);
14      setMunicipio(municipio);
15      setEstado(estado);
16      setFuncao("propor leis municipais em benefício da popula
17  }
```

Um novo erro ocorreu, conforme esperado. O atributo nome, presente no arquivo *Politico.h*, está com o método de acesso privado, sendo acessível somente dentro da classe Politico. No arquivo *Politico.h*, altere na linha 11 o método de acesso para protegido conforme o código a seguir. Em seguida, recompile o programa.

```
11  protected:
12      string nome;
13      string partido;
14      string estado;
15      string funcao;
```

Desta vez, funcionou! O programa compila.O método protegido permite acesso dentro da própria classe e dentro das classes derivadas. Fora da classe, este acesso não é permitido. Altere o arquivo *Main.cpp* conforme mostrado a seguir. Após alterar, tente recompilar.

```
24      cout << endl << vered->nome << endl;
```

O compilador não permite tal operação. Erro ao compilar. Outro aspecto interessante quando falamos de encapsulamento: como a classe-base compartilha seus métodos/atributos com as classes derivadas? Observe a linha 11 no arquivo *Vereadores.h*

```
11  class Vereador : public Politico {
```

O modo como está compartilhada a classe-base com a classe derivada Vereador é o modo público. O reflexo deste compartilhamento pode ser observado na tabela 7.1.

Tabela 7.1 – Modos de acesso aos métodos/atributos com as classes derivadas

Quando a classe derivada herdar a classe-base no modo	Como o acesso será quando o método/atributo estiver no modo		
	Private	Protected	Public
Private	Dentro da classe-base	Dentro da classe-base Dentro da classe derivada	Dentro da classe-base Dentro da classe derivada
Protected	Dentro da classe-base	Dentro da classe-base Dentro da classe derivada	Dentro da classe-base Dentro da classe derivada
Public	Dentro da classe-base	Dentro da classe-base Dentro da classe derivada	Dentro da classe-base Dentro da classe derivada Fora da classe derivada

Sistema político brasileiro em Python

Nesta implementação do aplicativo Político, iremos desenvolver as classes derivadas para presidente, senador, deputado federal, governador, deputado estadual, prefeito e vereador. Por uma questão de cultura, Python não restringe o acesso aos métodos e atributos das classes-base. Uma boa prática é utilizar o underline (_) em cada atributo "privado".

```
01  #!/usr/bin/python3
02  # coding: utf-8
03  """Classe Politico """
04
05  class Politico():
06      """ Classe Politico """
07
08      def __init__(self):
09          """ Construtor da classe Politico """
10          self.__nome = ""
11          self.__partido = ""
12          self.__estado = ""
13          self.__funcao = ""
14
15      def set_nome(self, nome):
16          """ Setar o nome do político """
17          if type(nome) == str:
18              self.__nome = nome
19
20      def get_nome(self):
21          """ Retorna o nome do político """
22          return self.__nome
23
24      def set_partido(self, partido):
25          """ Setar o partido do político """
26          if type(partido) == str:
27              self.__partido = partido
28
29      def get_partido(self):
30          """ Retorna o partido do político """
31          return self.__partido
```

```
32
33      def set_estado(self, estado):
34          """ Setar o estado (UF) do político """
35          if type(estado) == str:
36              self.__estado = estado
37
38      def get_estado(self):
39          """ Retorna o estado (UF) do político """
40          return self.__estado
41
42      def set_funcao(self, funcao):
43          """ Setar a funcao do político """
44          if type(funcao) == str:
45              self.__funcao = funcao
46
47      def get_funcao(self):
48          """ Retorna a funcao do político """
49          return self.__funcao
50
51      def apresentacao(self):
52          """ Realiza a apresentação do político """
53          print ('Olá, sou ' + self.get_nome())
54          print ('Meu partido é ' + self.get_partido())
```

Entre parênteses na linha 7, a classe Presidente herda a classe Politico.

```
01  #!/usr/bin/python3
02  # coding: utf-8
03  """Classe Presidente """
04
05  from Politico import Politico
06
07  class Presidente(Politico):
08      """ Classe Presidente """
09
10      def __init__(self, nome, partido):
11          """ Construtor da classe Presidente """
12          Politico.__init__(self)
13          self.set_nome(nome)
14          self.set_partido(partido)
15          self.set_estado("")
16          self.set_funcao("administrar bem os impostos federais em
```

Na classe Senador, conforme código a seguir, observamos na linha 18 uma forma de invocar um método da classe-base.

```
01  #!/usr/bin/python3
02  # coding: utf-8
03  """Classe Senador """
04
05  from Politico import Politico
06
07  class Senador(Politico):
```

```
08      """ Classe Senador """
09
10      def __init__(self, nome, partido, estado):
11          """ Construtor da classe Senador"""
12          Politico.__init__(self)
13          self.set_nome(nome)
14          self.set_partido(partido)
15          self.set_estado(estado)
16          self.set_funcao("propor no Senado leis em benefício da p
17
18      def apresentacao(self):
19          Politico.apresentacao(self)
20          print ('sou senador ')
21          print ('Minha função é ' + self.get_funcao())
22          print ('Fui eleito por ' + self.get_estado())
23          print ('============================')
```

As classes DeputadoFederal e DeputadoEstadual propositalmente nos mostram outra forma de invocar o método da classe-base. Observe como no método da apresentação dessas duas classes (linha 18) este outro método ocorre.

```
01  #!/usr/bin/python3
02  # coding: utf-8
03  """Classe DeputadoFederal """
04
05  from Politico import Politico
06
07  class DeputadoFederal(Politico):
08      """ Classe Deputado Federal """
09
10      def __init__(self, nome, partido, estado):
11          """ Construtor da classe DeputadoFederal """
12          Politico.__init__(self)
13          self.set_nome(nome)
14          self.set_partido(partido)
15          self.set_estado(estado)
16          self.set_funcao("propor na Câmara leis federais em benef
17
18      def apresentacao(self):
19          super(DeputadoFederal, self).apresentacao()
20          print ('sou deputado federal')
21          print ('Minha função é ' + self.get_funcao())
22          print ('Fui eleito por ' + self.get_estado())
23          print ('============================')
```

Como ocorreu na classe DeputadoFederal, o método apresentacao da classe DeputadoEstadual invoca o método da classe-base utilizando a palavra reservada super (linha 20).

```
01  #!/usr/bin/python3
02  # coding: utf-8
03  """Classe DeputadoEstadual """
04
05  from Politico import Politico
```

```
06
07
08  class DeputadoEstadual(Politico):
09      """ Classe Deputado Estadual """
10
11      def __init__(self, nome, partido, estado):
12          """ Construtor da classe DeputadoEstadual """
13          Politico.__init__(self)
14          self.set_nome(nome)
15          self.set_partido(partido)
16          self.set_estado(estado)
17          self.set_funcao("propor as leis estaduais de interesse d
18
19      def apresentacao(self):
20          super(DeputadoEstadual, self).apresentacao()
21          print ('sou deputado estadual')
22          print ('Minha função é ' + self.get_funcao())
23          print ('Fui eleito por ' + self.get_estado())
24          print ('============================')
25
```

O representante do poder executivo na esfera estadual também faz herança da classe Politico.

```
01  #!/usr/bin/python3
02  # coding: utf-8
03  """Classe Governador """
04
05  from Politico import Politico
06
07  class Governador(Politico):
08      """ Classe Senador """
09
10      def __init__(self, nome, partido, estado):
11          """ Construtor da classe Governador"""
12          Politico.__init__(self)
13          self.set_nome(nome)
14          self.set_partido(partido)
15          self.set_estado(estado)
16          self.set_funcao("administrar bem o ICMS para o bem do es
17
18      def apresentacao(self):
19          Politico.apresentacao(self)
20          print ('sou Governador ')
21          print ('Minha função é ' + self.get_funcao())
22          print ('Fui eleito por ' + self.get_estado())
23          print ('============================')
```

As classes Prefeito e Vereador requerem um atributo extra: municipio. No construtor dessas duas classes, esse atributo extra é observado (linha 16).

```
01  #!/usr/bin/python3
02  # coding: utf-8
03  """Classe Prefeito """
```

```
04
05  from Politico import Politico
06
07  class Prefeito(Politico):
08      """ Classe Prefeito """
09
10      def __init__(self, nome, partido, municipio, estado):
11          """ Construtor da classe Prefeito"""
12          Politico.__init__(self)
13          self.set_nome(nome)
14          self.set_partido(partido)
15          self.set_estado(estado)
16          self.__municipio = municipio
17          self.set_funcao("administrar o IPTU visando o melhor par
18
19      def set_municipio(self, municipio):
20          """ Setar o municipio do político. """
21          if type(municipio) == str:
22              self.__municipio = municipio
23
24      def get_municipio(self):
25          """ Retorna o nome municipio do político. """
26          return self.__municipio
27
28      def apresentacao(self):
29          Politico.apresentacao(self)
30          print ('sou prefeito: ' + self.get_municipio() + '/' + s
31          print ('Minha função é ' + self.get_funcao())
32          print ('Fui eleito por ' + self.get_estado())
33          print ('============================')
```

Na classe Vereador, observamos o atributo municipio também no construtor (linha 16).

```
01  #!/usr/bin/python3
02  # coding: utf-8
03  """Classe Vereador """
04
05  from Politico import Politico
06
07  class Vereador(Politico):
08      """ Classe Senador """
09
10      def __init__(self, nome, partido, municipio, estado):
11          """ Construtor da classe Vereador"""
12          Politico.__init__(self)
13          self.set_nome(nome)
14          self.set_partido(partido)
15          self.set_estado(estado)
16          self.__municipio = municipio
17          self.set_funcao("propor leis municipais em benefício da
18
```

```
19      def set_municipio(self, municipio):
20          """ Setar o municipio do político. """
21          if type(municipio) == str:
22              self.__municipio = municipio
23
24      def get_municipio(self):
25          """ Retorna o nome municipio do político. """
26          return self.__municipio
27
28      def apresentacao(self):
29          Politico.apresentacao(self)
30          print ('sou vereador: ' + self.get_municipio() + '/' + s
31          print ('Minha função é ' + self.get_funcao())
32          print ('Fui eleito por ' + self.get_estado())
33          print ('============================')
```

Após definir as classes, no arquivo *main.py* as importamos (linhas 5-11) e instanciamos os objetos (linhas 13-19):

```
01  #!/usr/bin/python3
02  # coding: utf-8
03  """ Exemplo do uso de herança. """
04
05  from Presidente import Presidente
06  from Senador import Senador
07  from DeputadoFederal import DeputadoFederal
08  from Governador import Governador
09  from DeputadoEstadual import DeputadoEstadual
10  from Prefeito import Prefeito
11  from Vereador import Vereador
12
13  PRES = Presidente("Frank Underwood","Partido House of Cards")
14  SEN = Senador("Michael Kern","Partido House of Cards","US")
15  DEP_F = DeputadoFederal("Diogo Fraga","Partido da Tropa II","RJ"
16  GOV = Governador("Jim Matthews","Partido House of Cards","Pensil
17  DEP_E = DeputadoEstadual("Fortunato","Partido da Tropa II","RJ")
18  PREF = Prefeito("Odorico Paraguaçu","Partido do Povo","Sucupira"
19  VERED = Vereador("Doroteia","Partido do Povo","Sucupira","BA")
20
21  PRES.apresentacao()
22  SEN.apresentacao()
23  DEP_F.apresentacao()
24  GOV.apresentacao()
25  DEP_E.apresentacao()
26  PREF.apresentacao()
27  VERED.apresentacao()
```

A saída da execução do código em Python pode ser observada a seguir:

```
Olá, sou Frank Underwood
Meu partido é Partido House of Cards
Olá, sou Michael Kern
Meu partido é Partido House of Cards
```

```
sou senador
Minha função é propor no Senado leis em benefício da população.
Fui eleito por US
============================
Olá, sou Diogo Fraga
Meu partido é Partido da Tropa II
sou deputado federal
Minha função é propor na Câmara leis federais em benefício da po
Fui eleito por RJ
============================
Olá, sou Jim Matthews
Meu partido é Partido House of Cards
sou Governador
Minha função é administrar bem o ICMS para o bem do estado.
Fui eleito por Pensilvânia
============================
Olá, sou Fortunato
Meu partido é Partido da Tropa II
sou deputado estadual
Minha função é propor as leis estaduais de interesse da populaçã
Fui eleito por RJ
============================
Olá, sou Odorico Paraguaçu
Meu partido é Partido do Povo
sou prefeito: Sucupira/BA
Minha função é administrar o IPTU visando o melhor para a cidade
Fui eleito por BA
============================
Olá, sou Doroteia
Meu partido é Partido do Povo
sou vereador: Sucupira/BA
Minha função é propor leis municipais em benefício da população.
Fui eleito por BA
============================
```

Síntese

Neste capítulo aprendemos como ocorre o relacionamento por herança. Herança é o mecanismo por meio do qual classes mais específicas incorporam a estrutura e o comportamento de classes mais gerais. Analisamos como os modos de acesso encapsulam os atributos e métodos em C++ e o conceito de sobreposição de métodos. Observamos como implementar herança em C++ e Python.

Tanto agregação e composição, vistas no capítulo anterior, quanto a herança estão relacionadas a uma das maiores vantagens em se adotar orientação a objetos: reúso de código.

CAPÍTULO 8

Herança múltipla

“Com grandes poderes vêm grandes responsabilidades.”

STAN LEE, ESCRITOR AMERICANO

O que iremos aprender?

- Herança múltipla.
- Programando “Os Vingadores”.

Herança múltipla é um conceito relativamente fácil, caso você tenha entendido herança simples. Entretanto, é um recurso que deve ser utilizado com cautela. Algumas linguagens orientada a objetos não a implementam. Talvez por isso seu professor de Java nunca tenha lhe apresentado este recurso. Quando se deriva duas ou mais classes ao mesmo tempo, estamos diante de um exemplo de herança múltipla.

Os Vingadores em C++

Você já assistiu ao filme *Os Vingadores*? Na formação do filme, temos os seguintes heróis: Hulk, Homem de Ferro, Thor, Capitão América, Gavião Arqueiro e Viúva Negra. Todos os heróis são pessoas. Alguns têm superforça, outros voam. Outros são apenas pessoas. A seguir, veremos as classes-base utilizadas por nossos heróis.

Vamos implementar em C++ apenas o Hulk e o Homem de Ferro. Inicialmente, criaremos a classe-base Pessoa.

```
01  /*
02  Programa:   Vingadores
03  Arquivo:    Pessoa.h
04  */
05  #ifndef PESSOA_H
06  #define PESSOA_H
07
08  using namespace std;
09
10  class Pessoa {
11  private:
12   string nome = "NÃO REVELADO";
13   string ocupacao = "NÃO REVELADO";
14   string sexo = "masculino";
15   float peso;
16   float altura;
17
18
19  public:
20   Pessoa();
21   ~Pessoa();
22
23   void setNome(string);
24   string getNome();
```

```
25
26   void setOcupacao(string);
27   string getOcupacao();
28
29   void setPeso(float);
30   float getPeso();
31
32   void setAltura(float);
33   float getAltura();
34
35   void getDados();
36  };
37
38  #endif
```

O arquivo *Pessoa.cpp* é apresentado a seguir.

```
01  /*
02  Programa:   Vingadores
03  Arquivo:    Pessoa.cpp
04  */
05  #include <stdio.h>
06  #include <stdlib.h>
07  #include <iostream>
08  #include "../includes/Pessoa.h"
09
10  using namespace std;
11
12  Pessoa::Pessoa() {
13   cout << "Construtor Pessoa." <<  endl;
14  }
15
16  Pessoa::~Pessoa() {
17   cout << "Destrutor Pessoa." <<  endl;
18  }
19
20  void Pessoa::setNome(string nome) {
21   this->nome = nome;
22  }
23
24  string Pessoa::getNome() {
25   return this->nome;
26  }
27
28  void Pessoa::setOcupacao(string ocupacao) {
29   this->ocupacao = ocupacao;
30  }
31
32  string Pessoa::getOcupacao() {
33   return this->ocupacao;
34  }
```

```
35
36  void Pessoa::setPeso(float peso) {
37   this->peso = peso;
38  }
39
40  float Pessoa::getPeso() {
41   return this->peso;
42  }
43
44  void Pessoa::setAltura(float altura) {
45   this->altura = altura;
46  }
47
48  float Pessoa::getAltura() {
49   return this->altura;
50  }
51
52  void Pessoa::getDados() {
53   cout << "Nome:  " << this->nome << endl;
54   cout << "Sexo:  " << this->sexo << endl;
55   cout << "Ocupação:  " << this->ocupacao << endl;
56   cout << "Peso:  " << this->peso << endl;
57   cout << "Altura:  " << this->altura << endl;
58  }
```

O poder da superforça está na classe-base SuperForca, codificado nos arquivos *SuperForca.h* e *SuperForca.cpp.*

```
01  /*
02  Programa:   Vingadores
03  Arquivo:    SuperForca.h
04  */
05  #ifndef SUPERFORCA_H
06  #define SUPERFORCA_H
07
08  using namespace std;
09
10  class SuperForca {
11  private:
12   float forca = 100;
13
14  public:
15   SuperForca();
16   ~SuperForca();
17
18   void setForca(float);
19   float getForca();
20
21  };
22
23  #endif
```

A seguir, o arquivo *SuperForca.cpp*:

```
01  /*
02  Programa:   Vingadores
03  Arquivo:    SuperForca.cpp
04  */
05  #include <stdio.h>
06  #include <stdlib.h>
07  #include <iostream>
08  #include "../includes/SuperForca.h"
09
10  using namespace std;
11
12  SuperForca::SuperForca() {
13   cout << "Construtor SuperForca." <<  endl;
14  }
15
16  SuperForca::~SuperForca() {
17   cout << "Destrutor SuperForca." <<  endl;
18  }
19
20  void SuperForca::setForca(float forca) {
21   this->forca = forca;
22  }
23
24  float SuperForca::getForca() {
25   return this->forca;
26  }
27
```

Outro poder importante para os nossos heróis é o poder de voar, codificado na classe-base Voar, codificada nos arquivos *Voar.h* e *Voar.cpp*.

```
01  /*
02  Programa:   Vingadores
03  Arquivo:    Voar.h
04  */
05  #ifndef VOAR_H
06  #define VOAR_H
07
08  using namespace std;
09
10  class Voar {
11  private:
12   float altitude = 0;
13
14  public:
15   Voar();
16   ~Voar();
17
18   void setAltitude(float);
19   float getAltitude();
20
```

```
21  };
22
23  #endif
```

A seguir, o arquivo *Voar.cpp*:

```
01  /*
02  Programa:   Vingadores
03  Arquivo:    Voar.cpp
04  */
05  #include <iostream>
06  #include "../includes/Voar.h"
07
08  using namespace std;
09
10  Voar::Voar() {
11   cout << "Construtor Voar." <<  endl;
12  }
13
14  Voar::~Voar() {
15   cout << "Destrutor Voar." <<  endl;
16  }
17
18  void Voar::setAltitude(float altitude) {
19   this->altitude = altitude;
20  }
21
22  float Voar::getAltitude() {
23   return this->altitude;
24  }
```

Definidas as classes-base, como é o poder de nossos heróis? O Hulk é uma pessoa que tem superforça. Mas essa superforça não está disponível a todo momento. Apenas quando ele fica irritado. Ao se acalmar, ele volta ao estado normal.

```
01  /*
02  Programa:   Vingadores
03  Arquivo:    Hulk.h
04  */
05  #ifndef HULK_H
06  #define HULK_H
07
08  #include "Pessoa.h"
09  #include "SuperForca.h"
10
11  using namespace std;
12
13  class Hulk : private Pessoa, private SuperForca {
14  private:
15
16  public:
17   Hulk();
18   ~Hulk();
```

```
19
20   void ficarNervoso();
21   void vaiPescar();
22
23   void getDados();
24  };
25
26  #endif
```

Observe no arquivo *Hulk.h* como a herança múltipla ocorre (linha 13). A seguir, o arquivo *Hulk.cpp*.

```
01  /*
02  Programa:   Vingadores
03  Arquivo:    Hulk.cpp
04  */
05  #include <stdio.h>
06  #include <stdlib.h>
07  #include <iostream>
08  #include "../includes/Hulk.h"
09
10  using namespace std;
11
12  Hulk::Hulk() {
13   this->setNome("Bruce Banner");
14   Pessoa::setOcupacao(
"cientista formado como físico nuclear especialista em raio gama");
15   this->setAltura(1.70);
16   Pessoa::setPeso(75);
17   SuperForca::setForca(30);
18  }
19
20  Hulk::~Hulk() {
21   cout << "Destrutor do Hulk." << endl;
22  }
23
24  void Hulk::ficarNervoso() {
25   cout << "\n\n Hulk esmaga !!!\n\n" << endl;
26   this->setNome("HUUUULLLKKK !!!!");
27   this->setOcupacao("esmagador profissional");
28   this->setAltura(3.20);
29   this->setPeso(500);
30  // SuperForca::setForca(10000);
31   this->setForca(10000);
32  }
33
34  void Hulk::vaiPescar() {
35   cout << "\n\n Hulk Calmo....\n\n" << endl;
36   this->setNome("Bruce Banner");
37   Pessoa::setOcupacao(
"cientista formado como físico nuclear especialista em raio gama");
38   this->setAltura(1.70);
```

```
39   this->setPeso(75);
40   this->setForca(30);
41  }
42
43  void Hulk::getDados() {
44   Pessoa::getDados();
45  // cout << "Forca:" << this->forca << endl;
46   cout << "Forca:" << this->getForca() << endl;
47  }
```

O homem de ferro é uma pessoa que tem superforça e voa caso a armadura esteja acionada. No filme, o Capitão América perguntou: “Um homem grande com armadura. Sem isso o que sobra?”. E tomou a resposta: “Um gênio, bilionário, filantropo.” No arquivo *IronMan.h* observamos a herança múltipla na linha 14.

```
01  /*
02  Programa:   Vingadores
03  Arquivo:    IronMan.h
04  */
05  #ifndef IRONMAN_H
06  #define IRONMAN_H
07
08  #include "Pessoa.h"
09  #include "SuperForca.h"
10  #include "Voar.h"
11
12  using namespace std;
13
14  class IronMan : private Pessoa, public SuperForca, public Voar {
15  private:
16   bool armaduraAtivada = false;
17  public:
18   IronMan();
19   ~IronMan();
20
21   void ativarArmadura();
22   void desativarArmadura();
23   bool getStatusArmadura();
24  };
25
26  #endif
```

A seguir, o arquivo *IronMan.cpp*.

```
01  /*
02  Programa:   Vingadores
03  Arquivo:    IronMan.cpp
04  */
05  #include <stdio.h>
06  #include <stdlib.h>
07  #include <iostream>
08  #include "../includes/IronMan.h"
09
```

```
10  using namespace std;
11
12  IronMan::IronMan() {
13   this->setNome("Antony Stark");
14   Pessoa::setOcupacao(
"cientista, empresário, gênio bilionário, playboy e filantrópico");
15   this->setAltura(1.75);
16   Pessoa::setPeso(80);
17  }
18
19  IronMan::~IronMan() {
20   cout << "Destrutor do Homem de Ferro." << endl;
21  }
22
23  void IronMan::ativarArmadura() {
24   this->armaduraAtivada = true;
25
cout << "\n\n Javis diz: Pronto para o combate, senhor.\n\n" << endl
26  }
27
28  void IronMan::desativarArmadura() {
29
cout << "\n\n Javis diz: Desativando armadura de combate.\n\n" << en
30   SuperForca::setForca(0);
31   Voar::setAltitude(0);
32   this->armaduraAtivada = false;
33  }
34
35  bool IronMan::getStatusArmadura() {
36   return this->armaduraAtivada;
37  }
```

No código *Main.cpp* observamos a criação dos objetos Hulk e IronMan.

```
01  /*
02  Programa:   Vingadores
03  Arquivo:    IronMan.cpp
04  */
05  #include <stdio.h>
06  #include <stdlib.h>
07  #include <iostream>
08  #include "../includes/IronMan.h"
09
10  using namespace std;
11
12  IronMan::IronMan() {
13   this->setNome("Antony Stark");
14   Pessoa::setOcupacao(
"cientista, empresário, gênio bilionário, playboy e filantrópico");
15   this->setAltura(1.75);
16   Pessoa::setPeso(80);
```

```
17  }
18
19  IronMan::~IronMan() {
20   cout << "Destrutor do Homem de Ferro." << endl;
21  }
22
23  void IronMan::ativarArmadura() {
24   this->armaduraAtivada = true;
25
cout << "\n\n Javis diz: Pronto para o combate, senhor.\n\n" << endl
26  }
27
28  void IronMan::desativarArmadura() {
29
cout << "\n\n Javis diz: Desativando armadura de combate.\n\n" << en
30   SuperForca::setForca(0);
31   Voar::setAltitude(0);
32   this->armaduraAtivada = false;
33  }
34
35  bool IronMan::getStatusArmadura() {
36   return this->armaduraAtivada;
37  }
```

A execução do programa é apresentada a seguir.

```
    Construtor Pessoa.
    Construtor SuperForca.
    Nome:  Bruce Banner
    Sexo:  masculino
                                                          Ocupação:
cientista formado como físico nuclear especialista em raio gama
    Peso:  75
    Altura:  1.7
    Forca:30
     Hulk esmaga !!!
    Nome:  HUUUULLLKKK !!!!
    Sexo:  masculino
    Ocupação:  esmagador profissional
    Peso:  500
    Altura:  3.2
    Forca:10000
     Hulk Calmo....
    Nome:  Bruce Banner
    Sexo:  masculino
                                                          Ocupação:
cientista formado como físico nuclear especialista em raio gama
    Peso:  75
    Altura:  1.7
    Forca:30
    Destrutor do Hulk.
```

```
Destrutor SuperForca.
Destrutor Pessoa.
*************************
Construtor Pessoa.
Construtor SuperForca.
Construtor Voar.
 Javis diz: Pronto para o combate, senhor.
 Altitude atual: 100
 SuperForça atual: 1500
 Javis diz: Desativando armadura de combate.
 Altitude atual: 0
 SuperForça atual: 0
Destrutor do Homem de Ferro.
Destrutor Voar.
Destrutor SuperForca.
Destrutor Pessoa.
```

Os Vingadores em Python

Hora de implementarmos os Vingadores em Python. Dividimos as classes em dois arquivos contendo as classes *poderes.py* e *herois.py*. Teremos um arquivo com a função principal (*main.py*).

```
01  #!/usr/bin/python3
02  # coding: utf-8
03  """ Módulo Poderes """
04
05
06  class SuperForca:
07      def __init__(self):
08          print("Tenho poder de Super Força.")
09          self.forca = 0
10
11      def get_forca(self):
12          return self.forca
13
14      def set_forca(self, forca):
15          self.forca = forca
16
17      def get_dados(self):
18          print("Forca:" + str(self.get_forca()))
19
20
21  class Voar:
22      def __init__(self):
23          print("Consigo voar.")
24          self.altitude = 0
25
26      def get_altitude(self):
27          return self.altitude
28
29      def set_altitude(self, altitude):
```

```
30          self.altitude = altitude
31
32      def get_dados(self):
33          if self.get_altitude() > 0:
34              print("Voo em:" + str(self.get_altitude()) + " metro
35
36
37  class Pessoa:
38      def __init__(self):
39          print("Sou uma pessoa.")
40          self.nome = "NÃO REVELADO"
41          self.ocupacao = "NÃO REVELADO"
42          self.sexo = "masculino"
43          self.peso = 0
44          self.altura = 0
45
46      def get_nome(self):
47          return self.nome
48
49      def set_nome(self, nome):
50          self.nome = nome
51
52      def get_ocupacao(self):
53          return self.ocupacao
54
55      def set_ocupacao(self, ocupacao):
56          self.ocupacao = ocupacao
57
58      def get_sexo(self):
59          return self.sexo
60
61      def set_sexo(self, sexo):
62          self.sexo = sexo
63
64      def get_peso(self):
65          return self.peso
66
67      def set_peso(self, peso):
68          self.peso = peso
69
70      def get_altura(self):
71          return self.altura
72
73      def set_altura(self, altura):
74          self.altura = altura
75
76      def get_dados(self):
77          print("Nome:" + self.get_nome())
78          print("Sexo:" + self.get_sexo())
79          print("Função:" + self.get_ocupacao())
```

```
80          print("Peso:" + str(self.get_peso()))
81          print("Altura:" + str(self.get_altura()))
```

O código a seguir encontra-se no arquivo *herois.py*.

```
01  #!/usr/bin/python3
02  # coding: utf-8
03  """ Módulo Heróis. """
04
05  from poderes import *
06
07
08  class Hulk(Pessoa, SuperForca):
09      def __init__(self):
10          Pessoa.__init__(self)
11          SuperForca.__init__(self)
12          self.vai_pescar()
13
14      def ficar_nervoso(self):
15          print("Hulk ESMAGA !!!")
16          self.set_nome("HUUUULLLKKK !!!!")
17          self.set_ocupacao("esmagador profissional")
18          self.set_altura(3.20)
19          self.set_peso(500)
20          self.set_forca(10000)
21
22      def vai_pescar(self):
23          print(" Hulk Calmo...")
24          self.set_nome("Bruce Banner")
25          self.set_ocupacao("cientista físico nuclear especialista
26          self.set_altura(1.70)
27          self.set_peso(75)
28          self.set_forca(30)
29
30      def get_dados(self):
31          print("")
32          Pessoa.get_dados(self)
33          SuperForca.get_dados(self)
34          print("")
35
36
37  class IronMan(Pessoa, SuperForca, Voar):
38      def __init__(self):
39          Pessoa.__init__(self)
40          SuperForca.__init__(self)
41          Voar.__init__(self)
42          self.armaduraAtivada = False
43          self.set_nome("Antony Stark")
44          self.set_ocupacao("empresário, gênio bilionário, playboy
45          self.set_altura(1.75)
46          self.set_peso(80)
```

```
47          self.set_forca(30)
48
49      def ativar_armadura(self):
50          self.armaduraAtivada = True
51          print("\nJavis diz: Pronto para o combate, senhor.")
52          self.set_forca(200)
53
54      def desativar_armadura(self):
55          print("\nJavis diz: Desativando armadura de combate.")
56          self.set_forca(30)
57          self.set_altitude(0)
58          self.armaduraAtivada = False
59
60      def set_altitude(self, altitude):
61          if self.armaduraAtivada:
62              self.altitude = altitude
63          else:
64              print("Javis diz: Não consigo voar sem a armadura, s
65
66      def get_dados(self):
67          print("")
68          if self.armaduraAtivada:
69              print("Armadura ativada: modo Homem de Ferro.")
70          Pessoa.get_dados(self)
71          SuperForca.get_dados(self)
72          Voar.get_dados(self)
73          print("")
74
75
76  class Thor(Pessoa, SuperForca, Voar):
77      def __init__(self):
78          Pessoa.__init__(self)
79          SuperForca.__init__(self)
80          self.set_nome("Donald Blake")
81          self.set_ocupacao("médico e deus do Trovão")
82          self.set_altura(1.80)
83          self.set_peso(85)
84          self.set_forca(1000)
85
86      def get_dados(self):
87          print("")
88          print("Sou um Deus, filho de Odin, mas só...")
89          Pessoa.get_dados(self)
90          SuperForca.get_dados(self)
91          print("")
92
93
94  class CapitaoAmerica(Pessoa, SuperForca):
95      def __init__(self):
96          Pessoa.__init__(self)
```

```
97          SuperForca.__init__(self)
98          self.set_nome("Steven Grant 'Steve' Rogers")
99          self.set_ocupacao("soldado geneticamente modificado")
100          self.set_altura(1.80)
101          self.set_peso(85)
102          self.set_forca(1000)
103
104      def get_dados(self):
105          print("")
106          Pessoa.get_dados(self)
107          SuperForca.get_dados(self)
108          print("")
109
110
111  class ViuvaNegra(Pessoa):
112      def __init__(self):
113          Pessoa.__init__(self)
114          self.set_nome("Natasha Alianovna Romanoff")
115          self.set_ocupacao("agente secreta da KGB")
116          self.set_sexo("feminino")
117          self.set_altura(1.70)
118          self.set_peso(60)
119
120      def get_dados(self):
121          print("")
122          Pessoa.get_dados(self)
123          print("")
124
125
126  class Arqueiro(Pessoa):
127      def __init__(self):
128          Pessoa.__init__(self)
129          self.set_nome("Clinton Francis 'Clint' Barton")
130          self.set_ocupacao("arqueiro por hobby e profissão")
131          self.set_altura(1.75)
132          self.set_peso(65)
133
134      def get_dados(self):
135          print("")
136          Pessoa.get_dados(self)
137          print("")
```

O código *main.py* é apresentado a seguir.

```
01  #!/usr/bin/python3
02  # coding: utf-8
03  """ Exemplo de herança múltipla """
04  from herois import *
05
06  if __name__ == "__main__":
07      MarkRuffalo = Hulk()
```

```
08      MarkRuffalo.get_dados()
09      MarkRuffalo.ficar_nervoso()
10      MarkRuffalo.get_dados()
11      MarkRuffalo.vai_pescar()
12      MarkRuffalo.get_dados()
13      print('--------------------------------')
14      RobertDowneyJr = IronMan()
15      RobertDowneyJr.get_dados()
16      RobertDowneyJr.set_altitude(150)
17      RobertDowneyJr.ativar_armadura()
18      RobertDowneyJr.set_altitude(150)
19      RobertDowneyJr.get_dados()
20      RobertDowneyJr.desativar_armadura()
21      RobertDowneyJr.get_dados()
22      print('--------------------------------')
23      ChrisHemsworth = Thor()
24      ChrisHemsworth.get_dados()
25      print('--------------------------------')
26      ScarlettJohansson = ViuvaNegra()
27      ScarlettJohansson.get_dados()
28      print('--------------------------------')
29      ChrisEvans = CapitaoAmerica()
30      ChrisEvans.get_dados()
31      print('--------------------------------')
32      JeremyRenner = Arqueiro()
33      JeremyRenner.get_dados()
34      print('--------------------------------')
35  #    print(Hulk.__mro__)
```

A execução do programa Vingadores é mostrada a seguir.

```
Sou uma pessoa.
Tenho poder de Superforça.
 Hulk Calmo...
Nome: Bruce Banner
Sexo: masculino
Função: cientista físico nuclear especialista em raio gama
Peso: 75
Altura: 1.7
Forca: 30
Hulk ESMAGA !!!
Nome: HUUUULLLKKK !!!!
Sexo: masculino
Função: esmagador profissional
Peso: 500
Altura: 3.2
Forca: 10000
 Hulk Calmo...
Nome: Bruce Banner
Sexo: masculino
Função: cientista físico nuclear especialista em raio gama
```

```
Peso: 75
Altura: 1.7
Forca: 30
---------------------------------
Sou uma pessoa.
Tenho poder de Superforça.
Consigo voar.
Nome: Antony Stark
Sexo: masculino
Função: empresário, gênio bilionário, playboy e filantropo
Peso: 80
Altura: 1.75
Forca: 30
Javis diz: Não consigo voar sem a armadura, senhor.
Javis diz: Pronto para o combate, senhor.
 Armadura ativada: modo Homem de Ferro.
Nome: Antony Stark
Sexo: masculino
Função: empresário, gênio bilionário, playboy e filantropo
Peso: 80
Altura: 1.75
Forca: 200
Voo em: 150 metros.
Javis diz: Desativando armadura de combate.
Nome: Antony Stark
Sexo: masculino
Função: empresário, gênio bilionário, playboy e filantropo
Peso: 80
Altura: 1.75
Forca: 30
---------------------------------
Sou uma pessoa.
Tenho poder de Superforça.
Sou um Deus, filho de Odin, mas só...
Nome: Donald Blake
Sexo: masculino
Função: médico e deus do Trovão
Peso: 85
Altura: 1.8
Forca: 1000
---------------------------------
Sou uma pessoa.
Nome: Natasha Alianovna Romanoff
Sexo: feminino
Função: agente secreta da KGB
Peso: 60
Altura: 1.7
---------------------------------
Sou uma pessoa.
Tenho poder de Superforça.
```

```
Nome: Steven Grant 'Steve' Rogers
Sexo: masculino
Função: soldado geneticamente modificado
Peso: 85
Altura: 1.8
Forca: 1000
--------------------------------
Sou uma pessoa.
Nome: Clinton Francis 'Clint' Barton
Sexo: masculino
Função: arqueiro por hobby e profissão
Peso: 65
Altura: 1.75
--------------------------------
```

As classes Hulk, Pessoa e Superforca contêm o método get_dados. No arquivo *main.py* (linhas 8, 10 e 12) é invocado o método da classe Hulk. Caso a classe não use esse método, é invocado get_dados da classe Pessoa, seguido pela classe Superforca.

Podemos observar pelo atributo de classe __mro__ (Method Resolution Order) qual a ordem de resolução de métodos. Descomente a linha 35 do arquivo *main.py* e observe o resultado.

```
(<class 'herois.Hulk'>, <class 'poderes.Pessoa'>, <class 'poderes.Su
```

Síntese

Neste capítulo nos aprofundamos no tópico herança. Herança múltipla é o mecanismo por meio do qual classes mais específicas incorporam a estrutura e o comportamento de duas ou mais classes mais gerais. Embora seja um mecanismo interessante, deve ser utilizado com cautela.

CAPÍTULO 9

Classes abstratas e polimorfismo

“Não tente entortar a colher, isso é impossível.
Em vez disso, apenas tente ver a verdade... não existe colher.”

GAROTO DA COLHER PARA O NEO, MATRIX (1999)

O que iremos aprender?

- Classes abstratas
- Métodos virtuais
- Polimorfismo

Nos capítulos anteriores aprendemos como funciona herança. Neste ponto do livro está claro para você o que é classe-base, classe derivada e o que são as formas de encapsulamento. Imagine a seguinte situação: você quer definir um método na classe-base, mas não quer implementá-lo ou quer que a implementação varie conforme a classe derivada mude. Este tipo de método é conhecido como método virtual. Quando se forçam as classes derivadas a reimplementar este método, as chamamos de funções virtuais puras. Essas classes que fazem uso de funções virtuais e funções virtuais puras são classes que não devem ser instanciadas. A função da classe abstrata é servir de base para as classes derivadas.

Hackeando a Matrix

No filme *Matrix* (1999), Morpheus (Laurence Fishburne) tenta explicar a Neo (Keanu Reeves) o que é a Matrix:

“A Matrix é um sistema, Neo. E esse sistema é nosso inimigo. Quando você está dentro e olha ao redor, o que você vê? Empresários, professores, advogados e carpinteiros. As mentes das próprias pessoas que estamos tentando salvar. Mas, até conseguirmos, essas pessoas ainda serão uma parte desse sistema, o que as tornam nosso inimigo. Você tem que entender, a maioria dessas pessoas não está pronta para ser desligada do sistema, muitas delas estão tão acostumadas, tão desesperadamente dependentes do sistema, que vão lutar para protegê-lo.”

Morpheus não era muito bom em orientação a objetos. Se soubesse, iria explicar da seguinte forma: “Neo, a Matrix é um grande sistema orientado a objetos, em que as pessoas são objetos de classes que herdam da classe abstrata Agente.” Vamos aos códigos da nossa Matrix:

```
01  /*
02  Programa:   Matrix
03  Arquivo:    matrix.cpp
04  */
05  #include <iostream>
06
07  using namespace std;
08
09  class Agente
10  {
11      private:
12          string nome;
13          bool modo_agente = false;
14          bool get_mode(){
15              return this->modo_agente;
```

```
16          }
17      protected:
18          string profissao;
19      public:
20          void set_nome(string nome)
21          {
22              this->nome = nome;
23          }
24          void mode_on() {
25              this->modo_agente = true;
26          }
27          virtual void apresentacao()
28          {
29              if(this->get_mode()){
30                  cout << "AGENTE SMITH !!!!" << endl;
31              } else {
32                       cout << "\n Olá ! Meu nome é " << this-
>nome << endl;
33              }
34          }
35  //        virtual void set_profissao() = 0;
36  };
37  class Empresario: public Agente {
38      private:
39          string empresa;
40      public:
41          Empresario(string nome, string empresa) {
42              set_nome(nome);
43              this->empresa = empresa;
44  //            this->set_profissao();
45          }
46          void apresentacao()
47          {
48              Agente::apresentacao();
49                       cout << "sou executivo da " << this-
>empresa << endl;
50          }
51      //                    void   set_profissao(){        this-
>profissao = "Empresário";    }
52  };
53  class Professor : public Agente
54  {
55      private:
56          string escola;
57      public:
58          Professor(string nome, string escola) {
59               set_nome(nome);
60              this->escola = escola;
61  //            this->set_profissao();
62          }
```

```
63          void apresentacao()
64          {
65              Agente::apresentacao();
66                      cout << "Leciono na escola " << this-
>escola << endl;
67          }
68      //                      void   set_profissao(){        this-
>profissao = "Professor";     }
69  };
70  class Advogado : public Agente
71  {
72      private:
73          string OAB;
74      public:
75          Advogado(string nome, string OAB) {
76              set_nome(nome);
77              this->OAB = OAB;
78  //            this->set_profissao();
79          }
80          void apresentacao()
81          {
82              Agente::apresentacao();
83                  cout << "sou advogado e meu OAB é: " << this-
>OAB << endl;
84          }
85       //                           void   set_profissao(){   this-
>profissao = "Advogado";        }
86  };
```

No código *matrix.cpp* podemos observar a classe Agente (linhas 9-36). Também observamos as classes derivadas Empresario (linhas 37-52), Professor (linhas 53-69) e Advogado (linhas 70-86), todos herdando a classe Agente. O método apresentacao() (linha 27-34) na classe Agente é um método virtual.

```
01  /*
02  Programa:   Matrix
03  Arquivo:    main.cpp
04  */
05  #include <iostream>
06  #include "matrix.cpp"
07  #define  N  5
08
09  using namespace std;
10
11  int main()
12  {
13      Agente *Pessoas[N];
14      Professor *Prof;
15      Empresario *Empr;
16      Advogado *Adv;
17      string nome,parametro2;
18
19      int contador = 0;
```

```
20      int i = 0;
21      char profissao;
22      for(i=0; i<N; i++){
23          cout  << "Escolha a profissão: "<< endl;
24          cout  << "1) Empresário   ";
25          cout  << "2) Professor    ";
26          cout  << "3) Advogado "<< endl;
27          cin >> profissao;
28          if(profissao == '1'){
29              cout  << "Digite o nome do empresário(a): ";
30              cin >> nome;
31              cout  << "Digite o nome da empresa: ";
32              cin >> parametro2;
33              Empr = new Empresario(nome,parametro2);
34              Pessoas[contador++] = Empr;
35          } else if(profissao == '2'){
36              cout  << "Digite o nome do professor(a): ";
37              cin >> nome;
38              cout  << "Digite o nome da escola: ";
39              cin >> parametro2;
40              Prof = new Professor(nome,parametro2);
41              Pessoas[contador++] = Prof;
42          } else {
43              cout  << "Digite o nome do advogado(a): ";
44              cin >> nome;
45              cout  << "Digite o número OAB: ";
46              cin >> parametro2;
47              Adv = new Advogado(nome,parametro2);
48              Pessoas[contador++] = Adv;
49          }
50      }
51      cout << "================================" << endl;
52      for(i=0; i<N; i++) {
53          Pessoas[i]->apresentacao();
54      }
55
56      Pessoas[2]->mode_on();
57      Pessoas[4]->mode_on();
58
59      cout << "================================" << endl;
60      for(i=0; i<N; i++) {
61          Pessoas[i]->apresentacao();
62      }
63  }
```

No código *main.cpp*, explicamos o que é a Matrix. Um vetor de N (definimos N na linha 7) Pessoas do tipo Agente. Se Morpheus soubesse OO, entenderia que pessoa é um (relacionamento de herança) agente. Entre as linhas 22 e 50, um laço for é utilizado para inserir dados no vetor de pessoas. Entre as linhas 52 e 54, essas pessoas se apresentam. Nas linhas 56 e 57 ocorre o problema: essas pessoas invocam o método mode_on() que ativa o agente da classe-base. Vamos à execução do código:

```
    Escolha a profissão:
```

```
1) Empresário   2) Professor   3) Advogado
1
Digite o nome do empresário(a): Bruce
Digite o nome da empresa: Wayne
Escolha a profissão:
1) Empresário   2) Professor   3) Advogado
2
Digite o nome do professor(a): Orlando
Digite o nome da escola: Uniararas
Escolha a profissão:
1) Empresário   2) Professor   3) Advogado
3
Digite o nome do advogado(a): Saul
Digite o número OAB: 17171
Escolha a profissão:
1) Empresário   2) Professor   3) Advogado
1
Digite o nome do empresário(a): SeuBarriga
Digite o nome da empresa: autonomo
Escolha a profissão:
1) Empresário   2) Professor   3) Advogado
2
Digite o nome do professor(a): Girafales
Digite o nome da escola: daVilla
================================
Olá ! Meu nome é Bruce
sou executivo da Wayne
Olá ! Meu nome é Orlando
Leciono na escola Uniararas
Olá ! Meu nome é Saul
sou advogado e meu OAB é: 17171
Olá ! Meu nome é SeuBarriga
sou executivo da autonomo
Olá ! Meu nome é Girafales
Leciono na escola daVilla
================================
Olá ! Meu nome é Bruce
sou executivo da Wayne
Olá ! Meu nome é Orlando
Leciono na escola Uniararas
AGENTE SMITH !!!!
sou advogado e meu OAB é: 17171
Olá ! Meu nome é SeuBarriga
sou executivo da autonomo
AGENTE SMITH !!!!
Leciono na escola daVilla
```

Em nossa matrix, o Saul (do seriado Better Call Saul) e o professor Girafales (do seriado Chaves) tornaram-se o temido agente Smith.

Convido você a hackear a matrix, a fim de aprimorar seu conhecimento. Remova a palavra virtual no

código *matrix.cpp* (linha 27). Recompile e rode novamente o programa. A seguir, a saída apenas do trecho final da execução.

```
Olá ! Meu nome é Bruce
Olá ! Meu nome é Orlando
Olá ! Meu nome é Saul
Olá ! Meu nome é SeuBarriga
Olá ! Meu nome é Girafalles
================================
Olá ! Meu nome é Bruce
Olá ! Meu nome é Orlando
AGENTE SMITH !!!!
Olá ! Meu nome é SeuBarriga
AGENTE SMITH !!!!
```

Sem a palavra reservada virtual, o método apresentacao() invocado foi o método da classe-base. A palavra reservada virtual é importante. Ela torna o método virtual e nos diz que podem existir novas definições desse método em classe derivadas. Reinsira a palavra virtual no início do método e insira na linha 35 do código *matrix.cpp* conforme a seguir:

```
35          virtual void set_profissao() = 0;
```

Tente recompilar o programa. Não vai dar certo. Este método que acabamos de inserir é um método virtual puro. Nenhuma implementação é feita na classe-base, e o método é igualado a zero.

Desta forma, todas as classes derivadas são obrigadas a reimplementar o método set_profissao(). A seguir, as mudanças necessárias e em quais linhas podem ser feitas a sobreposição desse método virtual e a invocação desse método no construtor de cada classe.

```
    44              this->set_profissao();
    ...
          51                          void   set_profissao()   {this-
>profissao = "Empresário";}
    ...
    61              this->set_profissao();
    ...
          68                          void   set_profissao()   {this-
>profissao = "Professor";}
    ...
    78              this->set_profissao();
    ...
          85                          void   set_profissao()   {this-
>profissao = "Advogado";}
```

Ao fazer uso de métodos virtuais, as classes derivadas podem reimplementar o método. Ao fazer uso de métodos virtuais puros, as classes derivadas obrigatoriamente precisam reimplementar o método.

Hackeando a Matrix com Python

Quando Morpheus está preso no prédio, o agente Smith faz uma revelação interessante: “Você sabia que o primeiro Matrix foi projetado para ser um mundo humano perfeito? Onde não teria nenhum sofrimento, onde todos seriam felizes. Foi um desastre. Ninguém iria aceitar o programa. Colheitas inteiras foram perdidas. Alguns acreditavam que faltava a linguagem de programação para descrever o seu mundo perfeito.” Morpheus poderia responder: “Vocês tentaram fazer em Python?”. Nem Morpheus nem os agentes tentaram. Nós, sim.

Com uso do módulo ABC (Abstract Base Classes) e do decorador @abc.abstractmethod (linhas15 e 20), o construtor e o método apresentacao() precisam ser sobrescritos nas classes derivadas (Executivo, Professor

e Advogado).

```
01  #!/usr/bin/python3
02  # coding: utf-8
03  """ Módulo Matrix """
04  import abc
05
06
07  class Agente(metaclass=abc.ABCMeta):
08      """ Classe abstrata Agente """
09      modo_agente = False
10
11      def mode_on(self):
12          """ Método para ativação do agente """
13          self.modo_agente = True
14
15      @abc.abstractmethod
16      def __init__(self, nome):
17          """ Construtor abstrado da classe-base """
18          self.nome = nome
19
20      @abc.abstractmethod
21      def apresentacao(self):
22          """ Método de apresentacao """
23          if self.modo_agente:
24              print ('Oi, sou o Agente Smith')
25          else:
26              print ('Olá ! Meu nome é ' + str(self.nome))
27
28
29  class Executivo(Agente):
30      """ Classe Executivo """
31      def __init__(self, nome, empresa):
32          Agente.__init__(self, nome)
33          self.empresa = empresa
34          self.profissao = "Executivo"
35
36      def apresentacao(self):
37          Agente.apresentacao(self)
38          print ('Sou executivo(a) na empresa ' + str(self.empresa
39
40
41  class Professor(Agente):
42      """ Classe Professor """
43      def __init__(self, nome, escola):
44          Agente.__init__(self, nome)
45          self.escola = escola
46          self.profissao = "Professor"
47
48      def apresentacao(self):
49          Agente.apresentacao(self)
```

```
50          print ('Sou professor(a) na escola ' + str(self.escola))
51
52
53  class Advogado(Agente):
54      """ Classe Advogado """
55      def __init__(self, nome, oab):
56          Agente.__init__(self, nome)
57          self.oab = oab
58          self.profissao = "Advogado"
59
60      def apresentacao(self):
61          Agente.apresentacao(self)
62          print ('Sou advogado(a). Meu OAB: ' + str(self.oab))
63
```

Nosso código *main.py* do sistema Matrix em Python:

```
01  #!/usr/bin/python3
02  # coding: utf-8
03  """ Exemplo do uso de classe abstrata """
04  from matrix import *
05
06  if __name__ == "__main__":
07      Pessoas = []
08      for i in range(5):
09          print ("Escolha : \n 1) Empresário  2) Professor  3) Adv
10          profissao = input()
11          if profissao == '1':
12              nome = input("Digite o nome: ")
13              empresa = input("Digite a empresa: ")
14              obj = Executivo(nome, empresa)
15          elif profissao == '2':
16              nome = input("Digite o nome do professor: ")
17              escola = input("Digite a escola: ")
18              obj = Professor(nome, escola)
19
20          else:
21              nome = input("Digite o nome: ")
22              oab = input("Digite o número OAB: ")
23              obj = Advogado(nome, oab)
24          Pessoas.append(obj)
25
26      for i in range(5):
27          Pessoas[i].apresentacao()
28
29      Pessoas[2].mode_on()
30      Pessoas[4].mode_on()
31      print('====================')
32      for i in range(5):
33          Pessoas[i].apresentacao()
34
```

A execução da matrix em Python é apresentada a seguir.

```
Escolha :
 1) Empresário   2) Professor   3) Advogado
1
Digite o nome: Bruce
Digite a empresa: Wayne
Escolha :
 1) Empresário   2) Professor   3) Advogado
2
Digite o nome do professor: Orlando
Digite a escola: Uniararas
Escolha :
 1) Empresário   2) Professor   3) Advogado
3
Digite o nome: Saul
Digite o número OAB: 17171
Escolha :
 1) Empresário   2) Professor   3) Advogado
1
Digite o nome: Seu Barriga
Digite a empresa: autonomo
Escolha :
 1) Empresário   2) Professor   3) Advogado
2
Digite o nome do professor: Girafalles
Digite a escola: da Villa
Olá ! Meu nome é Bruce
Sou executivo(a) na empresa Wayne
Olá ! Meu nome é Orlando
Sou professor(a) na escola Uniararas
Olá ! Meu nome é Saul
Sou advogado(a). Meu OAB: 17171
Olá ! Meu nome é Seu Barriga
Sou executivo(a) na empresa autonomo
Olá ! Meu nome é Girafalles
Sou professor(a) na escola da Villa
====================
Olá ! Meu nome é Bruce
Sou executivo(a) na empresa Wayne
Olá ! Meu nome é Orlando
Sou professor(a) na escola Uniararas
Oi, sou o Agente Smith
Sou advogado(a). Meu OAB: 17171
Olá ! Meu nome é Seu Barriga
Sou executivo(a) na empresa autonomo
Oi, sou o Agente Smith
Sou professor(a) na escola da Villa
```

No código matrix.py, comente as linhas 36 a 38, conforme mostrado a seguir.

```
36 #      def apresentacao(self):
```

```
37 #          Agente.apresentacao(self)
38 #          print ('Sou executivo(a) na empresa ' + str(self.e
```

Rode novamente a matrix em Python. Um erro é esperado assim que se cadastra o primeiro empresário:

```
Traceback (most recent call last):
  File "./main.py", line 14, in <module>
    obj = Executivo(nome, empresa)
TypeError: Can't instantiate abstract class Executivo with abstr
```

O erro ocorre porque não podemos instanciar uma classe que não implementa um método abstrato.

Polimorfismo

A palavra polimorfismo vem do grego (poli = muitas, morphos = formas). Assim, o que fizemos no programa Matrix é um exemplo de polimorfismo. A classe Agente, representada pelo objeto Pessoas, reproduziu o comportamento das classes derivadas. Conheci pessoas que consideravam polimorfismo apenas ter várias sobrecargas de um determinado método. Mas, afinal, o que é polimorfismo?

Luca Cardelli e Peter Wegner propuseram no artigo “On Understanding Types, Data Abstraction, and Polymorphism” (1985) quatro tipos de polimorfismo, conforme mostra a figura 9.1.

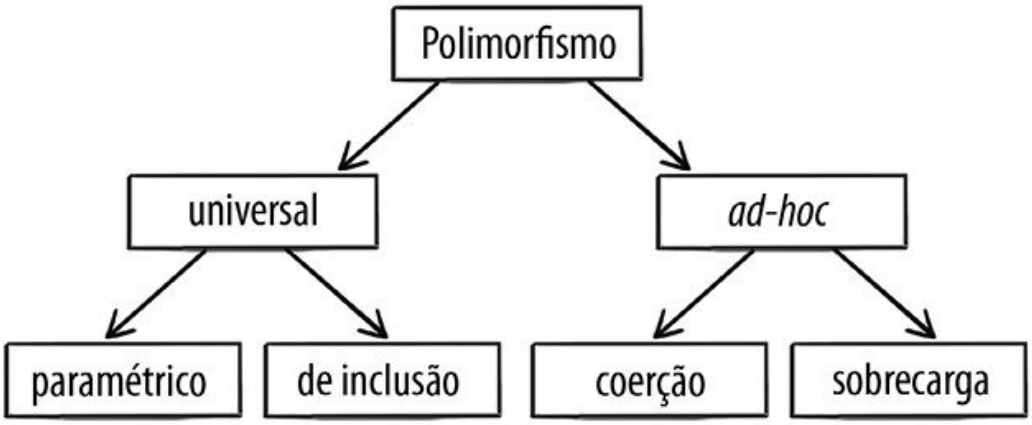


Figura 9.1 – Formas de polimorfismo.

No polimorfismo de inclusão, um objeto genérico pode fazer uso de um objeto mais específico posteriormente. Neste tipo de polimorfismo, os elementos dos subtipos (classes derivadas) são também elementos do supertipo (classe-base). Vamos ao código do programa Transformers.

```
01  /*
02  Programa:   Transformers
03  Arquivo:    inclusao.cpp
04  */
05  #include <iostream>
06
07  using namespace std;
08  class Transformers
09  {
10  public:
11      virtual void emite_som() {}
12      void ativar() {
13          this->transformado = true;
14      }
15      void desativar(){
16          this->transformado = false;
```

```
17      }
18      bool getStatusAtivado(){
19          return this->transformado;
20      }
21  protected:
22      bool transformado = false;
23  };
24
25  class Camaro: public Transformers
26  {
27  public:
28      void emite_som()
29      {
30          if (Transformers::getStatusAtivado()) {
31              cout << "sou o Bumbeblee !!"<< endl;
32          } else {
33              cout << "VRRRRRrrruummmmmm"<< endl;
34          }
35      }
36      void ativar() {
37          cout << "Transformando em Bumbeblee"<< endl;
38          Transformers::ativar();
39      }
40  };
41
42  class Caminhao: public Transformers
43  {
44  public:
45      void emite_som()
46      {
47          if (getStatusAtivado()) {
48              cout << "Sou o Optimus Prime !!"<< endl;
49          } else {
50              cout << "VRRRRRrrruummmmmm !!!"<< endl;
51          }
52      }
53  };
54
55  class Uno: public Transformers
56  {
57  public:
58      void emite_som()
59      {
60          if (getStatusAtivado()) {
61                  cout << "Sou uma churrasqueira George Foreman."
<< endl;
62              cout << "O que você esperava. Sou um transformers de
<< endl;
63          } else {
64                  cout << "sem ronco de motor... não liga... "
```

```
<< endl;
65          }
66      }
67  };
68
69  int main(void)
70  {
71      Transformers *generico;
72      Camaro carro;
73      Caminhao caminhao;
74      Uno carro2;
75
76      generico = &carro;
77      generico->emite_som();
78      generico->ativar();
79      generico->emite_som();
80      generico->desativar();
81      generico->emite_som();
82      cout << "\n ======================== \n";
83
84      generico = &caminhao;
85      generico->emite_som();
86      generico->ativar();
87      generico->emite_som();
88      generico->desativar();
89      generico->emite_som();
90      cout << "\n ======================== \n";
91
92      generico = &carro2;
93      generico->emite_som();
94      generico->ativar();
95      generico->emite_som();
96      generico->desativar();
97      generico->emite_som();
98
99      generico = NULL;
100      return 0;
101  }
```

No código *inclusao.cpp*, no programa Transformers, a variável generico é uma variável polimórfica de uma classe. Essa variável tem a capacidade de referenciar objetos da classe e objetos de qualquer subclasse. Temos a classe-base Transformers (linhas 8 a 23) e as classes derivadas Camaro (linhas 25-40), Caminhao (linhas 42-53) e Uno (linhas 55-67). No filme não tem o carro modelo Uno, mas em nosso código tem. A execução do programa Transformers é apresentada a seguir.

```
VRRRRRrrruummmmmm
sou o Bumbeblee !!
VRRRRRrrruummmmmm
 ========================
VRRRRRrrruummmmmm !!!
Sou o Optimus Prime !!
VRRRRRrrruummmmmm !!!
```

```
 ========================
sem ronco de motor... não liga...
Sou uma churrasqueira George Foreman.
O que você esperava. Sou um transformers de Fiat Uno ???
sem ronco de motor... não liga...
```

Na linha 71 temos um ponteiro da classe-base (generico), e a partir deste, consegue-se ser um Camaro para transformar-se no Bumbeblee, um caminhão para transformar-se no Optimus Prime e em um Uno para virar uma churrasqueira.

No polimorfismo paramétrico, como o nome propõe, os parâmetros são o foco. O C++ tem a ver com uso de templates. Se você vem do mundo Java, deve conhecer ou ter ouvido falar de generics. Vamos aos códigos do programa Parametrico:

```
01  /*
02  Programa:   Parametrico
03  Arquivo:    parametrico.cpp
04  */
05  #include <iostream>
06  using namespace std;
07
08  template <typename T>
09  void troca(T &a, T &b) {
10      cout << "Troca !" << endl;
11      T temp = a;
12      a = b;
13      b = temp;
14  }
15
16  int main() {
17  // Inteiros
18      int x1 = 10, x2 = 20;
19      cout << " x1 = " << x1  << endl << " x2 = " << x2  << endl;
20      troca(x1,x2);
21      cout << " x1 = " << x1  << endl << " x2 = " << x2  << endl;
22  // Float
23      float y1 = 10.4, y2 = 20.8;
24      cout << " y1 = " << y1  << endl << " y2 = " << y2  << endl;
25      troca(y1,y2);
26      cout << " y1 = " << y1  << endl << " y2 = " << y2  << endl;
27  // string
28      string s1, s2;
29      s1 = "Oi";
30      s2 = "Mundo!";
31      cout << " s1 = " << s1  << endl << " s2 = " << s2  << endl;
32      troca(s1,s2);
33      cout << " s1 = " << s1  << endl << " s2 = " << s2  << endl;
34      return 0;
35  }
```

No arquivo *parametrico.cpp*, caso a função troca fosse uma função monomórfica, ou seja, uma função que trabalha apenas com um tipo fixo, seria necessário reescrever várias funções iguais, mudando-se o tipo somente. Com o uso dos templates, temos uma função polimórfica, capaz de atender a tipos diferentes. No exemplo, executamos a troca com inteiros, números ponto flutuante e strings. O tipo T da função (linhas 09-

14) não é previamente conhecido. A execução deste programa é apresentada a seguir.

```
 x1 = 10
 x2 = 20
Troca !
 x1 = 20
 x2 = 10
 y1 = 10.4
 y2 = 20.8
Troca !
 y1 = 20.8
 y2 = 10.4
 s1 = Oi
 s2 = Mundo!
Troca !
 s1 = Mundo!
 s2 = Oi
```

É comum fazermos uso de conversões de tipos. Estas conversões podem ser explícitas ou implícitas para nós, usuários-programadores. Quando utilizamos recursos como *cast*, por exemplo, estamos fazendo uma conversão explícita. É de nosso conhecimento a mudança de tipo. Uma coerção é uma conversão de tipos implícita. Se você não ficar atento, a coerção ocorrerá sem você perceber.

Já a sobrecarga é a reescrita da função ou do método para atender a um tipo diferente. Qual a vantagem do polimorfismo paramétrico sobre a sobrecarga? Com o polimorfismo de sobrecarga, você precisará prever quais serão todos os tipos que serão atendidos. Ao fazer uso do polimorfismo paramétrico, o número variável de tipos atendido será potencialmente maior. No código *ad_hoc.cpp*, observamos o uso de polimorfismo por sobrecarga e por coerção.

```
01  /*
02  Programa:   AdHoc
03  Arquivo:    ad_hoc.cpp
04  */
05  #include <iostream>
06
07  using namespace std;
08
09  void troca(int &a, int &b) {
10   cout << "Troca de inteiros " << endl;
11   int temp = a;
12   a = b;
13   b = temp;
14  }
15  void troca(float &a, float &b) {
16   cout << "Troca de floats " << endl;
17   int temp = a;
18   a = b;
19   b = temp;
20  }
21  void imprimir(int a,int b) {
22   cout << "Número 1 =  " << a << endl;
23   cout << "Número 2 =  " << b << endl;
24  }
25
```

```
26  int main() {
27   int x1, x2;
28   x1 = 10;
29   x2 = 25;
30   cout << "Exemplo de CAST =  " << (float)x2/x1 << endl;
31
32   imprimir(x1,x2);
33   troca(x1,x2);
34   imprimir(x1,x2);
35
36   float y1, y2;
37   y1 = 10.4;
38   y2 = 20.8;
39   imprimir(y1,y2);
40   troca(y1,y2);
41   imprimir(y1,y2);
42
43   return 0;
44  }
```

No exemplo do código *ad_hoc.cpp*, a sobrecarga ocorre nas funções troca (linhas 9-14 e 15-20), prevendo-se atender aos tipos inteiro e float. Em uma eventual necessidade de atender o tipo string, por exemplo, uma nova função seria criada. A conversão explícita com uso do cast é apresentada na linha 30. Observe a função imprimir (linhas 21-24). Ao invocar essa função nas linhas 39 e 40, uma conversão implícita ocorre. Este é um exemplo de polimorfismo por coerção. A saída da execução do programa é apresentada a seguir.

```
Exemplo de CAST =  2.5
Número 1 =  10
Número 2 =  25
Troca de inteiros
Número 1 =  25
Número 2 =  10
Número 1 =  10
Número 2 =  20
Troca de floats
Número 1 =  20
Número 2 =  10
```

Outro exemplo de coerção pode ser observado no código *coercao.cpp*, apresentado a seguir.

```
01  /*
02  Programa:   Coercao
03  Arquivo:    coercao.cpp
04  */
05  #include <iostream>
06
07  int main() {
08   int i;
09   char c = 'a';
10   float x;
11   i = c;
12   c = i + 1;
13   x = i;
14   i = x / 7;
```

```
15   std::cout << c << std::endl;
16   std::cout << x << std::endl;
17   std::cout << i << std::endl;
18  }
```

A execução do programa coerção é apresentada a seguir. Observe como a variável c do tipo char fez uma operação de soma (linha 12) e apresentou o resultado “b” na linha 15. Como a variável i do tipo int recebeu um caractere (linha 11), como uma variável ponto-flutuante foi atribuída com um valor inteiro e como um ponto-flutuante dividido por um inteiro foi atribuído a uma variável inteiro (linha 14). Um programador desatento pode cometer um erro por não contar com a conversão implícita por coerção.

```
b
97
13
```

Polimorfismo em Python

Python tem uma forma de polimorfismo em que uma funcionalidade opera em qualquer objeto que implemente os métodos apropriadamente. Esse tipo de polimorfismo é conhecido como *duck typing*. Se caminha como pato e se comporta como pato, é pato! Vamos ao código para um melhor entendimento:

```
01  #!/usr/bin/python3
02  # coding: utf-8
03  """ Exemplo de DuckType """
04
05
06  class Pato:
07      def nadar(self):
08          print("quack, quack, quack")
09
10      def som(self):
11          print('QUAAACCK !!! ')
12
13
14  class PatinhoFeio:
15      def nadar(self):
16          print('Quãã...Quãã...')
17
18      def som(self):
19          print('Quãã... ')
20
21
22  class Pluto:
23      def som(self):
24          print(' Au Au Au Au !!!')
25
26
27  class Donald(Pato):
28      pass
29
```

Uma forma não muito polimórfica de fazer polimorfismo é apresentada a seguir. Checar se o objeto é uma instância de Pato (linha 8) permite encontrar os objetos Pato e Donald somente.

```
01  #!/usr/bin/python3
```

```
02  # coding: utf-8
03
04  from duckType import *
05
06  def encontrarPato(animal):
07      """Descobrir se o animal é pato ou não"""
08      if isinstance(animal, Pato):
09          animal.nadar()
10          animal.som()
11
12  if __name__ == "__main__":
13      Animais = {}
14      Animais['Pato'] = Pato()
15      Animais['Donald'] = Donald()
16      Animais['Pluto'] = Pluto()
17      Animais['PatinhoFeio'] = PatinhoFeio()
18
19      for chave, valor in Animais.items():
20          print ("Animal: " + chave)
21          encontrarPato(valor)
22          print ("")
```

A execução desta versão não tão polimórfica é apresentada a seguir.

```
    Animal: Donald
    quack, quack, quack
    QUAAACCK !!!
    Animal: Pluto
    Animal: Pato
    quack, quack, quack
    QUAAACCK !!!
    Animal: PatinhoFeio
```

Se você se lembrar da clássica história Patinho Feio, se surpreenderá ao descobrir que o patinho feio era um cisne. Mas falando-se em *duck-typing*, andou como pato, comportou-se como pato, é pato!

```
01  #!/usr/bin/python3
02  # coding: utf-8
03
04  from duckType import *
05
06  def encontrarPato(animal):
07      try:
08          animal.nadar()
09          animal.som()
10      except AttributeError as e:
11          print(e)
12
13  if __name__ == "__main__":
14      dict = {}
15      dict['Pato'] = Pato()
16      dict['Donald'] = Donald()
17      dict['Pluto'] = Pluto()
```

```
18      dict['PatinhoFeio'] = PatinhoFeio()
19
20      for chave, valor in dict.items():
21          print("Animal: " + chave)
22          encontrarPato(valor)
23          print("")
```

A execução é apresentada a seguir. A classe PatinhoFeio não é Pato ou uma classe que herde da classe Pato. Mas contém todos os métodos compatíveis com Pato: nadar() e som().

```
Animal: Donald
quack, quack, quack
QUAAACCK !!!
Animal: Pato
quack, quack, quack
QUAAACCK !!!
Animal: PatinhoFeio
Quãã...Quãã...
Quãã...
Animal: Pluto
'Pluto' object has no attribute 'nadar'
```

Síntese

Neste capítulo aprendemos como funcionam as classes abstratas. Uma classe abstrata tem pelo menos um método abstrato e não pode ser instanciada. Conversamos sobre os métodos virtuais e os métodos virtuais puros. Conhecemos os quatro tipos de polimorfismo, exemplificando seu uso, a linguagem C++. Conhecemos os tipos-patos (*duck types*) e seu uso em Python.

APÊNDICE A

Exercícios

“ Só mais um round!”
ROCKY BALBOA

O que iremos aprender?

- Taxonomia de Bloom
- Exercícios para fixação
- Projetos para fixação
- Correção dos exercícios

Taxonomia de Bloom

O psicólogo educacional Benjamin Samuel Bloom (1913-1999) deu uma enorme contribuição quanto à classificação dos objetivos educacionais e à teoria da aprendizagem de domínio. Ele defendeu que o saber acontece a partir da organização das capacidades em níveis cognitivos.

Segundo a taxonomia dos objetivos educacionais (popularmente conhecida como taxonomia de Bloom), as habilidades cognitivas tratam de conhecimento, compreensão e do pensar sobre um problema ou fato. Na tabela A.1 podemos ver os objetivos educacionais cognitivos, relacionados aos verbos, e uma descrição do objetivo educacional.

Tabela A.1 – A taxonomia de Bloom e os objetivos educacionais

Objetivos educacionais	Verbos	Descrição
Conhecimento	Citar Enunciar Definir Descrever Listar Indicar Relacionar	É o objetivo educacional mais elementar na área cognitiva. O aluno se limita a repetir os aspectos cognitivos de problemas
Compreensão	Apresentar Explicar Expor Focalizar Assinalar Concordar Discordar	O aluno, além de reproduzir o assunto da forma como aprendeu, deve demonstrar capacidade para explicar o significado do que aprendeu
Aplicação	Aplicar Elaborar Relacionar	O aluno, já sendo capaz de reproduzir e de explicar a ideia aprendida, aplica-a em situações novas e concretas
Análise	Analisar Examinar Estudar Resumir Classificar Identificar Distinguir Comparar	O aluno, além das capacidades adquiridas, demonstra ser capaz de decompor o assunto em partes e de estudar cada parte, de forma a ser a sua estrutura
Síntese	Sintetizar Integrar Propor Selecionar	Considerando que já atingiu os níveis anteriores, o aluno, agora, demonstra ter a capacidade de integrar as partes e atingir um novo todo
Avaliação	Avaliar Justificar Estimar Apreciar Julgar Interpretar Destacar (com juízo de valor)	É o objetivo educacional mais profundo da aprendizagem, pois nele o aluno demonstra sua capacidade de apresentar juízo de valor próprio sobre o assunto

Os exercícios e projetos deste capítulo têm como objetivo fortalecer o conhecimento sobre orientação a objetos. Cada exercício elaborado está relacionado a um objetivo educacional distinto. Conforme os objetivos educacionais forem se aprofundando, os desafios aumentarão.

Exercícios para fixação

1) Objetivo educacional: conhecimento

Preencha as lacunas:

a) Na orientação a objeto, as instâncias de uma classe são representadas por_____.

b) _____ é a capacidade do paradigma orientado a objetos de ocultar dados dentro de modelos, permitindo que somente operações especializadas ou dedicadas manipulem os dados ocultos.

c) _____ no contexto da programação orientada a objetos é o mecanismo que permite a uma classe (subclasse) estender outra classe (superclasse), de forma a aproveitar comportamentos (métodos) e variáveis (atributos), é denominado_____.

d) Os métodos de um objeto de uma classe são ativados por _____enviadas por objetos de outras classes.

2) Objetivo educacional: compreensão

Observe o código a seguir:

```
01  #!/usr/bin/python3
02  # coding: utf-8
03
04  A= [1,2,3,4]
05  print (A)
06
07  for i in A:
08      print (i + ' ')
09
```

A execução do código anterior apresenta o seguinte erro:

```
[1, 2, 3, 4]
Traceback (most recent call last):
  File "teste2.py", line 8, in <module>
    print (i + ' ')
TypeError: unsupported operand type(s) for +: 'int' and 'str
```

Sobre o código e o erro apresentado, considere as seguintes afirmações:

I. Python é uma linguagem de tipagem forte e não permite a concatenação de inteiro com string

II. Para o código executar perfeitamente, altere a linha 08 para

print (str(i) + ' ')

III. Para o código executar perfeitamente, altere a linha 08 para

print (' '+ i + ' ')

Podemos afirmar como verdadeiro:

a) I

b) II

c) III

d) I e II

e) todas as afirmações são verdadeiras

3) Objetivo educacional: aplicação

Em Python3, escreva uma classe Transporte, sem construtor, com um método chamado mostrar. O método mostrar irá imprimir na tela a mensagem “‘Transporte sendo apresentado”.

Crie uma classe Caminhao que herde a classe Transporte. A classe Caminhao precisa ter um construtor que receba o modelo do caminhão como parâmetro e o armazene. A classe Caminhao sobrescreve o método mostrar e imprime a mensagem “Caminhão modelo” e o nome do modelo em seguida. Exemplo de execução em modo interativo das classes desenvolvidas por você é apresentado a seguir:

```
>>> from transporte import *
>>> ford = Caminhao('Ford')
>>> ford.mostrar()
Caminhão modelo Ford
>>> VW = Caminhao('Volks')
>>> VW.mostrar()
Caminhão modelo Volks
>>>
```

4) Objetivo educacional: análise

Observe o código a seguir:

```
01  #!/usr/bin/python3
02  #coding: utf-8
03
04  class A:
05      a = 1
06
07  class B(A):
08      _c = 3
09
10      def __init__(self):
11          print (self.a)
12          print (self._c)
13
14  a = A()
15  if isinstance(a, B):
16      print ('a é instância da classe B ')
17
18  b = B()
19  if isinstance(b, B):
20      print ('b é instância da classe B ')
21
22  c = A()
23  if isinstance(c, A):
24      print ('c é instância da classe A ')
25
26  d = B()
27  if isinstance(d, A):
28      print ('d é instância da classe A ')
```

Ao atribuir privilégio de execução e executar o programa anterior:

```
$ chmod +x heranca.py
$ ./heranca.py
```

A saída esperada no terminal será:

a)

1

3

b é instância da classe B

c é instância da classe A

1

3

b)

1

3

b é instância da classe B

c é instância da classe A

1

3

d é instância da classe A

c)

b é instância da classe B

c é instância da classe A

d é instância da classe A

d)

1

3

1

3

e)

a é instância da classe B

b é instância da classe B

c é instância da classe A

d é instância da classe A

5) Objetivo educacional: síntese

Em um sistema para simular uma aula, pretende-se desenvolver as classes Pessoa, Professor, Aluno, Sala de Aula. Qual associação entre essas quatro classes é a mais apropriada? Justifique sua resposta.

6) Objetivo educacional: avaliação

Em 17 de outubro de 2013, um usuário da lista Python Brasil enviou um e-mail com o título "Herança múltipla: você já precisou disso em Python?". Diversas respostas foram dadas. Uma das últimas respostas foi feita pelo escritor Luciano Ramalho, autor do livro *Python Fluente*. Ele disse:

"... assim como C++, Python não tem uma maneira formal de definir interfaces como em Java, então em Python e C++ usam-se classes abstratas para definir interfaces, com a vantagem de que tais classes podem trazer alguns métodos já implementados. Este uso de classes abstratas para definir interfaces é um motivo para você usar herança múltipla, tanto que até mesmo em Java herança múltipla de interfaces é suportada.

Dito tudo isso, existe uma tendência forte de usar herança em demasia quando se deveria usar composição. Quando se usa menos herança, a tendência natural é também usar menos herança múltipla."

Fonte: *https://groups.google.com/forum/#!topic/python-brasil/0TsyRXV5Bhc*

Elabore um e-mail de resposta direcionada ao grupo Python-Brasil em que você contribua com a discussão, justificando o uso de herança múltipla ou de composição.

Projetos para fixação

Objetivo educacional: aplicação

Políticos (C++)

Utilize os códigos apresentados no capítulo 7 para este projeto. Político não trabalha de graça. Todos os cargos apresentados são remunerados. Entretanto, nosso código não previu este fato. Crie na classe-base um atributo salario, um método setSalario(float) e um método getSalario() retornando float. Faça uso do método setSalario no construtor das classes derivadas (o salário de cada cargo fica a seu critério) e do método getSalario() no método apresentacao().

Políticos (Python)

Com base no projeto anterior, vamos fazer o mesmo com o código Python. Crie na classe-base um atributo salário, um método setSalario(salario) e um método getSalario(). Faça uso do método setSalario no construtor das classes derivadas (o salário de cada cargo fica a seu critério) e do método getSalario() no método apresentacao().

Caixa acoplada (Python)

Refatorar é o processo de modificar um sistema de software para melhorar a estrutura interna. Com base no projeto apresentado no capítulo 6, vamos refatorar o código, visando tornar os objetos mais "independentes", enfatizando a troca de mensagens. Nesse projeto, devemos utilizar como base o diagrama de sequências mostrado na figura A.1.

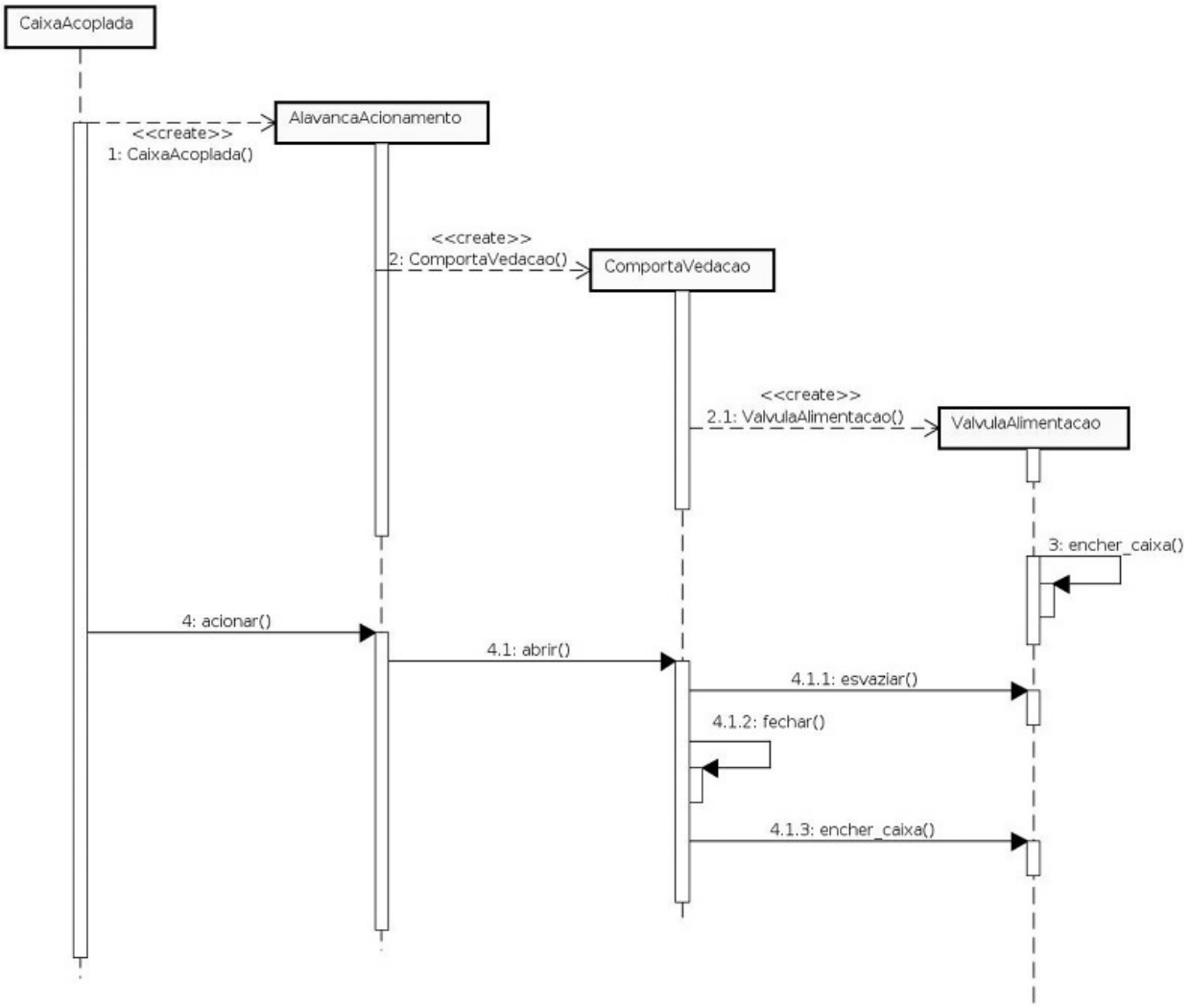


Figura A.1 – Novo diagrama de sequência.

Nenhuma alteração deve ser feita no código *main.py*. O processo de refatorar visa melhorias internas do sistema.

Objetivo educacional: análise

Fila de Pacientes 1 (C++)

Os códigos do projeto a seguir encontram-se no diretório */Projetos/pacientes/filaPacientes*.

Este projeto faz uso da biblioteca STL (Standard Template Library), biblioteca integrada à biblioteca-padrão da linguagem C++ para facilmente disponibilizar estruturas de dados básicas como lista, pilha, deque, set, multiset, map e multimap.

Neste projeto, um objeto do tipo FilaAtendimento gerencia internamente uma lista de objetos do tipo Pessoas. Esta classe contém os métodos para adicionar pacientes (incluir no fim da fila), atender (remover Pessoa do início da fila) e mostrar a fila atual. O trecho do código *main.cpp* é apresentado a seguir:

```
14      fila.mostrarFila();
15
16      fila.adicionar("Orlando", 34);
17      fila.adicionar("Maria", 64);
18      fila.adicionar("Ana", 33);
```

```
19      fila.adicionar("Lucas", 4);
20      fila.adicionar("Paulo", 65);
21      fila.adicionar("Ilma", 70);
22
23      fila.atender();
24      fila.atender();
25
26      fila.mostrarFila();
```

A saída da execução do projeto encontra-se a seguir:

```
Construtor da Fila
Lista vazia !

Atendendo Orlando - 34 anos
Atendendo Maria - 64 anos
  Os pacientes não atendidos ainda:
Nome: Ana  idade: 33
Nome: Lucas  idade: 4
Nome: Paulo  idade: 65
Nome: Ilma  idade: 70
Destrutor da Fila
Lista vazia !
```

Entretanto, para cumprir a lei do idoso, pessoas com mais de 60 anos devem ter atendimento prioritário. Assim, a classe FilaAtendimento deve gerenciar duas filas: uma para atendimento normal e outra para atendimentos prioritários. Faça as alterações necessárias, de forma que a saída do seu programa seja próxima à saída a seguir:

```
Construtor da Fila
Lista vazia !

Atendendo Maria - 64 anos  - atendimento preferencial.
Atendendo Paulo - 65 anos  - atendimento preferencial.
  Os pacientes com atendimento  preferencial não atendidos:
Nome: Ilma  idade: 70
  Os pacientes não atendidos ainda:
Nome: Orlando  idade: 34
Nome: Ana  idade: 33
Nome: Lucas  idade: 4
Destrutor da Fila
Lista vazia !
```

Objetivo educacional: síntese

Fila de Pacientes 2 (C++)

O sistema de filas de pacientes, contemplando a fila preferencial, está funcionando perfeitamente. Porém novos desafios serão propostos neste projeto.

1) Crie um destrutor para a classe Pessoa. No construtor, registre o tempo que o usuário entrou na fila e registre no destrutor o tempo em que este usuário foi atendido. O destrutor da classe Pessoa deverá imprimir na tela o tempo que o usuário ficou na fila.

DICA: pesquisar os headers <chrono> e <ctime> do C++.

2) No método adicionar(nome, idade) da classe FilaAtendimento, imprima uma mensagem alertando o usuário adicionado na fila.

3) Para tornar a simulação de fila mais real, os atendimentos realizados pela fila preferencial devem durar entre 1.500 e 3.500 milissegundos. Já os atendimentos na fila comum devem durar entre 1.000 e 2.500 milissegundos.

DICA: pesquisar os headers <chrono> e <thread> do C++.

4) No método mostrarFila() da classe FilaAtendimento, se a fila preferencial estiver vazia, imprima "Todos os pacientes preferenciais foram atendidos."; caso contrário, liste os pacientes preferenciais ainda na fila.

Se a fila convencional estiver vazia, imprima "Todos os pacientes foram atendidos."; caso contrário, listar os pacientes ainda na fila.

Caso todas as filas estejam vazias, imprima a mensagem "Todos os pacientes foram atendidos.".

Após todos os requisitos serem atendidos, altere o código-fonte *main.cpp* conforme o modelo a seguir.

```
01  #include <iostream>
02  #include "Pessoa.h"
03  #include "FilaAtendimento.h"
04
05  using namespace std;
06
07  int main()
08  {
09      FilaAtendimento fila;
10
11      fila.mostrarFila();
12      fila.adicionar("Orlando", 34);
13      fila.adicionar("Maria", 64);
14      fila.adicionar("Ana", 33);
15      fila.adicionar("Lucas", 4);
16      fila.adicionar("Paulo", 65);
17      fila.adicionar("Ilma", 70);
18
19      fila.atender();
20      fila.atender();
21      fila.atender();
22      fila.adicionar("Pedro", 74);
23      fila.atender();
24      fila.atender();
25      fila.adicionar("José Henrique", 32);
26      fila.atender();
27      fila.atender();
28
29      fila.mostrarFila();
30
31      return 0;
32  }
```

Com a alteração no código *main.cpp*, conforme apresentado anteriormente, espera-se obter uma saída semelhante a apresentada a seguir:

```
Construtor da Fila
Todos os pacientes foram atendidos.
 O paciente Orlando entrou na fila.
O paciente Maria entrou na fila prioritária.
```

```
O paciente Ana entrou na fila.
O paciente Lucas entrou na fila.
O paciente Paulo entrou na fila prioritária.
O paciente Ilma entrou na fila prioritária.
Atendendo Maria - 64 anos  - atendimento preferencial.
Tempo na fila: 1.99129 segundos
Atendendo Paulo - 65 anos  - atendimento preferencial.
Tempo na fila: 5.22844 segundos
Atendendo Ilma - 70 anos  - atendimento preferencial.
Tempo na fila: 6.73859 segundos
O paciente Pedro entrou na fila prioritária.
Atendendo Pedro - 74 anos  - atendimento preferencial.
Tempo na fila: 1.70512 segundos
Atendendo Orlando - 34 anos
Tempo na fila: 9.58203 segundos
O paciente José Henrique entrou na fila.
Atendendo Ana - 33 anos
Tempo na fila: 11.4542 segundos
Atendendo Lucas - 4 anos
Tempo na fila: 13.6713 segundos
Todos os pacientes preferenciais foram atendidos.

  Os pacientes não atendidos ainda:
Nome: José Henrique  idade: 32
Destrutor da Fila
Todos os pacientes foram atendidos.
```

Objetivo educacional: avaliação

Pizzaria (Python)

Uma determinada pizzaria tem um cardápio conforme o mostrado na tabela A.2.

Tabela A.2 – Sabores de uma pizzaria

Sabor da pizza	Ingredientes	Valores
Frango com Catupiry	Molho de tomate natural, frango desfiado ricamente temperado, legítimo catupiry, azeitonas graúdas e orégano. Se desejar, você pode trocar a massa tradicional pela massa 100% farinha integral.	P 34,80 M 40,80 G 47,40
Frango com Cheddar	Molho de tomate natural, frango desfiado ricamente temperado, cheddar, cobertura de batata palha, azeitonas graúdas e orégano. Se desejar, você pode trocar a massa tradicional pela massa 100% farinha integral.	P 35,20 M 41,20 G 47,80
Frango com Molho Barbecue	Molho de tomate natural, mussarela, frango desfiado ricamente temperado, milho verde, molho barbecue, azeitonas graúdas e orégano. Se desejar, você pode trocar a massa tradicional pela massa 100% farinha integral.	P 34,20 M 40,20 G 46,80
Peito de Peru	Molho de tomate natural, peito de peru, mussarela, azeitonas graúdas e orégano. Se desejar, você pode trocar a massa tradicional pela massa 100% farinha integral.	P 34,20 M 40,20 G 46,80
Peito de Peru com Catupiry	Molho de tomate natural, mussarela, peito de peru, legítimo catupiry, azeitonas graúdas e orégano. Se desejar, você pode trocar a massa tradicional pela massa 100% farinha integral.	P 37,20 M 43,20 G 49,80

Proponha a modelagem de uma classe-base que represente uma pizza e de classes derivadas. Desenvolva um sistema que interaja com o usuário via terminal, recebendo os pedidos das pizzas. Avalie a sua proposta com o código-modelo, destacando os pontos positivos e negativos de cada projeto desenvolvido.

Respostas

1)

a) objetos

b) encapsulamento

c) herança

d) mensagens

2) d) I e II.

3)

```
#!/usr/bin/python3
#coding: utf-8
class Transporte:
    def mostrar (self):
        print 'Transporte sendo apresentado'
class Caminhao(Transporte):
    def __init__(self,modelo):
        self.modelo = modelo
```

```
        def mostrar (self):
            print 'Caminhão modelo ' + str(self.modelo)
```

4) b

A função isinstance() verifica se um objeto é instância de determinada classe. No trecho a seguir, do código apresentado:

```
 19 if isinstance(b, B):
```

retorna verdadeiro caso o objeto b seja uma instância da classe B ou alguma subclasse de B.

Este exercício exige análise para identificar quem é subclasse de quem. Outro ponto a observar: ao instanciar a classe A, nenhuma mensagem é impressa na tela, mas ao instanciar a classe B, o objeto invoca o construtor (linhas 10 a 12) e imprime os números 1 e 3.

5) Quanto mais profundo for o objetivo educacional, maior dedicação exigirá do professor ao analisar as propostas e ideias do aluno. De modo genérico, espera-se nesta resposta a proposta de ter, na classe Sala de Aula, um objeto do tipo Professor e de 1 a N [1..N] objetos do tipo Aluno, sendo Professor e Aluno subclasses da classe Pessoa. Entretanto novas propostas podem surgir, como por exemplo a classe Professor ser composta de uma Pessoa e a classe Aluno ser composta de uma Pessoa.

6) Este objetivo educacional é o mais profundo, e isso exige mais preparo do aluno para elaborar uma resposta embasada no conhecimento teórico sobre o objeto de estudo (orientação a objetos) e, a partir daí, justificar o seu ponto de vista em relação a este. Para o professor que elabora este tipo de questão, o desafio na correção é grande, visto que o aluno pode argumentar a favor de um ponto de vista diferente do apresentado pelo docente.

Comentários sobre os projetos

Os projetos cujo objetivo educacional é a análise esperam de você mais do que relacionamento e associação dos conceitos. Espera-se que examine os códigos, procurando identificar onde deve ser alterado sem impacto nas demais partes do projeto.

Os projetos com o objetivo educacional de síntese, além de uma grande capacidade de associação e autodisciplina, requerem que se busquem novos conceitos em outras fontes, como outros livros e a internet. Nesta categoria de projeto, sua contribuição na resolução do problema proposto é fundamental.

Os projetos com o objetivo educacional de avaliação requerem sua avaliação e seu questionamento sobre o valor da sua proposta de código, identificando e classificando os pontos de melhorias.

Síntese

Neste capítulo conhecemos a taxonomia de Bloom e observamos como o tema orientação a objeto pode ser cobrado em provas/simulados/questões de concurso desde o objetivo educacional mais elementar (conhecimento) até o objetivo educacional mais profundo (avaliação). Aos leitores professores, uma provocação: as provas, listas e outras atividades (em sala e extras) estão alinhadas a quais objetivos educacionais?

Referências

Nesta seção, quero recomendar alguns bons sites e livros, divididos por assunto:

Sites sobre Python

• Grupo Python Brasil – Excelente referência para a linguagem no idioma português. Comunidade super-receptiva com os novos ingressantes.
https://groups.google.com/forum/#!forum/python-brasil

• Documentação da Linguagem na versão 3
https://docs.python.org/3/

• Code Cademy – Python
https://www.codecademy.com/learn/python

Livros sobre Python

Se você está começando na linguagem ou começando a programar, recomendo os seguintes livros:

• Pense em Python – Allen B. Downey
ISBN: 978-85-7522-508-0

• Introdução à Programação com Python – Nilo Ney Coutinho Menezes
ISBN: 978-85-7522-408-3

• Python para Desenvolvedores – Luiz Eduardo Borges
ISBN: 978-85-7522-405-2

Já para os experientes programadores Python, recomendo o livro *Python Fluente*. Na comunidade Python, o autor dispensa apresentações. Livro recheado de boas dicas para explorar ao máximo a linguagem.

• Python Fluente – Luciano Ramalho
ISBN: 978-85-7522-462-5

Sites sobre a linguagem C++

• Tutorial C++
http://www.cplusplus.com/doc/tutorial/

• Direto do site do criador da linguagem C++. Neste site há diversos links interessantes
http://www.stroustrup.com/C++.html

• Site sobre a padronização ISO C++ (ISO/IEC 14882:2014)
https://isocpp.org/std/the-standard

Livros sobre a linguagem C++

• A Tour of C++ – Bjarne Stroustrup
ISBN 978-0321958310

• Object Oriented Programming Using C++ – Joyce M. Farrell
ISBN 978-0760050446

• Thinking in C++ – Bruce Eckel
ISBN: 978-0139798092

• Thinking in C++, Volume 2: Practical Programming – Bruce Eckel
ISBN: 978-0130353139
• Mastering Object-Oriented Design in C++ – Cay S. Horstmann
ISBN: 978-0471594840

Outros livros sobre Orientação a Objetos

• UML 2 – Uma Abordagem Prática – Gilleanes T. A. Guedes
ISBN: 978-8575222812
• UML: guia do usuário – Grady Booch, James Rumbaugh, Ivar Jacobson
ISBN: 978-8535217841
• Utilizando UML e Padrões – Craig Larman
ISBN: 978-8560031528
• Conceitos de Linguagens de Programação – Robert W. Sebesta
ISBN: 978-8536301716

Objetivos educacionais

• A Taxonomy for Learning, Teaching, and Assessing:
A Revision of Bloom's Taxonomy of Educational Objectives, Abridged Edition
ISBN: 9780801319037


CONSTRUINDO UMA
LOJA VIRTUAL
A jornada de uma empreendedora
em seu primeiro negócio online
novatec
André Gugliotti

Construindo uma loja virtual

Gugliotti, André
9788575224953
217 páginas

Compre agora e leia

Juliana é uma arquiteta de sucesso, com estabilidade e um emprego com que sempre sonhou. No entanto, algo não vai bem e, em um dia de fúria, ela decide sacudir seu mundo, largar o emprego e abrir sua própria empresa.

No entanto, ela logo descobre que as coisas são mais complicadas do que parecem. Em sua jornada para ter sua primeira loja virtual, Juliana enfrenta diversos problemas e recebe ajuda de personagens emblemáticos como o Professor, um mestre na arte de empreender, e Pedro, o proprietário de uma loja virtual de calçados.

Em sua caminhada, ela aprende a planejar uma empresa, construir catálogos de produtos, cuidar das finanças e do marketing. Ela também descobre as matrioscas, bonecas russas de longa tradição e que serão vendidas em sua loja.

O que Juliana não sabe é que uma das matrioscas esconde um segredo que pode colocar sua vida e a de Pedro em perigo. Embarque nessa aventura, monte sua loja virtual e descubra o mistério da matriosca.

Compre agora e leia


JOVEM E
BEM-SUCEDIDO
Um guia para a realização
profissional e financeira
TRABALHO
DINHEIRO
ESTUDOS
novatec
Juliano Niederauer

Jovem e Bem-sucedido

Niederauer, Juliano
9788575225325
192 páginas

Compre agora e leia

Jovem e Bem-sucedido é um verdadeiro guia para quem deseja alcançar a realização profissional e a financeira o mais rápido possível. Repleto de dicas e histórias interessantes vivenciadas pelo autor, o livro desmistifica uma série de crenças relativas aos estudos, ao trabalho e ao dinheiro.

Tem como objetivo orientar o leitor a planejar sua vida desde cedo, possibilitando que se torne bem-sucedido em pouco tempo e consiga manter essa realização no decorrer dos anos. As três perspectivas abordadas são:

ESTUDOS: mostra que os estudos vão muito além da escola ou faculdade. Aborda as melhores práticas de estudo e a aquisição dos conhecimentos ideais e nos momentos certos.

TRABALHO: explica como você pode se tornar um profissional moderno, identificando oportunidades e aumentando cada vez mais suas fontes de renda. Fornece ainda dicas valiosas para desenvolver as habilidades mais valorizadas no mercado de trabalho.

DINHEIRO: explica como assumir o controle de suas finanças, para, então, começar a investir e multiplicar seu patrimônio. Apresenta estratégias de investimentos de acordo com o momento de vida de cada um, abordando as vantagens e desvantagens de cada tipo de investimento.

Jovem e Bem-sucedido apresenta ideias que o acompanharão a vida toda, realizando importantes mudanças no modo como você planeja estudar, trabalhar e lidar com o dinheiro.

Compre agora e leia


Tudo que você precisa saber para gerar negócios na maior rede social do mundo
Facebook
MARKETING
novatec
Camila Porto

Facebook Marketing

Porto, Camila
9788575224977
360 páginas

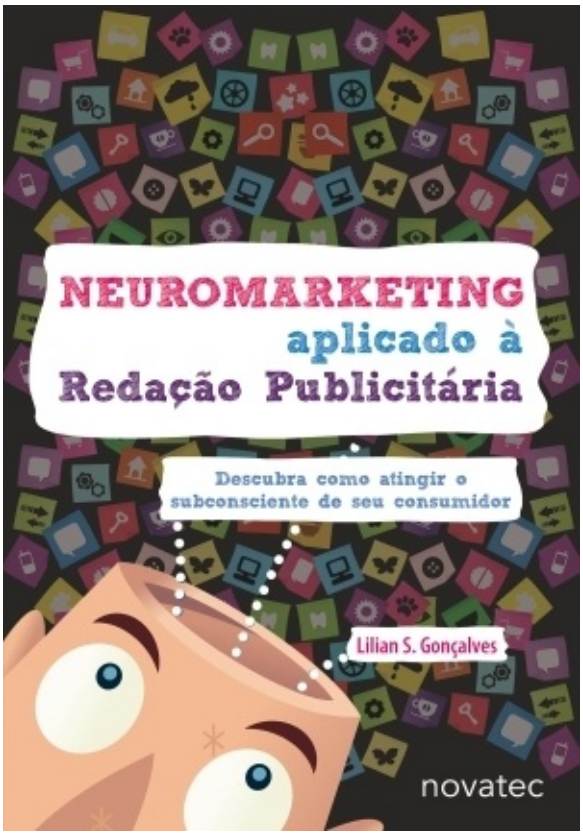
Compre agora e leia

Como a maior rede social do mundo pode se tornar uma peça-chave para o seu negócio? Como aumentar suas vendas, atrair e fidelizar clientes e conquistar fãs?

Estas e outras respostas você encontrará neste livro, além de tudo que precisa saber para utilizar o Facebook a fim de potencializar seu negócio e ter muito mais resultados. A partir de um conteúdo rico e objetivo, acompanhado de entrevistas exclusivas com profissionais renomados, você saberá como utilizar o Facebook desde a construção da sua estratégia, o que postar, como anunciar, como vender, como mensurar e como estar sempre onde seus clientes estão. Com um bilhão de usuários, o Facebook se tornou um dos canais mais importantes para os negócios, e você precisa saber o que ele é capaz de fazer pelo seu.

Neste livro você aprenderá:
- Como usar o Facebook para alavancar sua carreira;
- Como potencializar sua marca na maior rede social do mundo;
- Como produzir conteúdo e gerar engajamento;
- Como usar os anúncios no Facebook e conseguir mais visibilidade;
- Como integrar o Facebook ao seu site, e-commerce, blog de forma estratégica para vender mais;
- Como saber se suas ações estão gerando resultados, a partir da mensuração da sua presença no Facebook;
- Como gerenciar crises e transformá-las em oportunidades para o seu negócio.

Compre agora e leia


NEUROMARKETING
aplicado à
Redação Publicitária
Descubra como atingir o subconsciente de seu consumidor
Lilian S. Gonçalves
novatec

Neuromarketing Aplicado à Redação Publicitária

Gonçalves, Lilian S.
9788575224991
192 páginas

Compre agora e leia

O neuromarketing é uma nova visão para a publicidade mundial e pode ser a chave para uma campanha de sucesso.

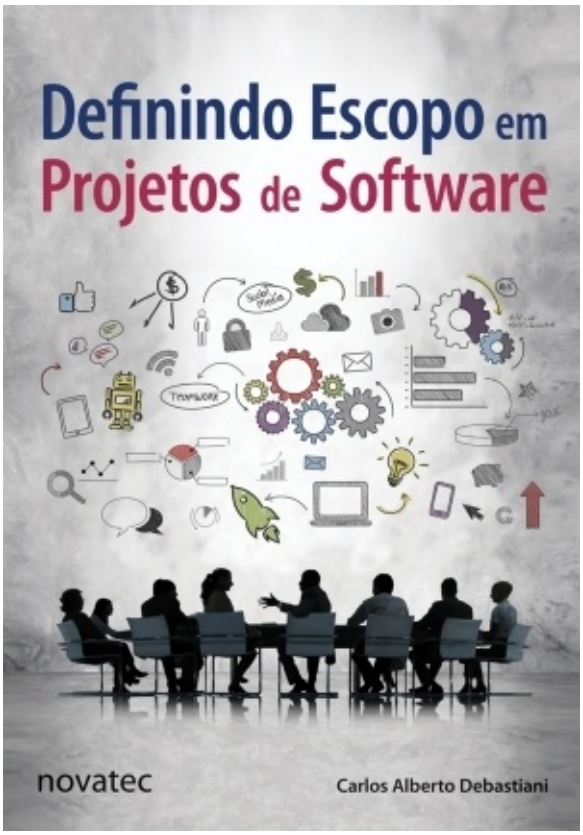
A versatilidade proporcionada pelo casamento entre as tradicionais pesquisas de mercado e as descobertas da neurociência se mostra um prato cheio para os redatores de plantão.

Esta obra reúne as principais descobertas feitas pelo neuromarketing nas últimas décadas e suas possíveis aplicações como argumentação para os anúncios publicitários.

Mais do que atingir seu público-alvo, chegou a hora de desenvolver campanhas de marketing com foco no subconsciente de seu consumidor, a fim de ampliar sua lista de clientes fidelizados.

Bom humor, criatividade, técnica e ciência estão de mãos dadas rumo a seu sucesso.

Compre agora e leia


Definindo Escopo em
Projetos de Software
novatec
Carlos Alberto Debastiani

Definindo Escopo em Projetos de Software

Debastiani, Carlos Alberto
9788575224960
144 páginas

Compre agora e leia

Definindo Escopo em Projetos de Software é uma obra que pretende tratar, de forma clara e direta, a definição de escopo como o fator mais influente no sucesso dos projetos de desenvolvimento de sistemas, uma vez que exerce forte impacto sobre seus custos. Abrange diversas áreas do conhecimento ligadas ao tema, abordando desde questões teóricas como a normatização e a definição das características de engenharia de software, até questões práticas como métodos para coleta de requisitos e ferramentas para desenho e projeto de soluções sistêmicas.

Utilizando uma linguagem acessível, diversas ilustrações e citações de casos vividos em sua própria experiência profissional, o autor explora, de forma abrangente, os detalhes que envolvem a definição de escopo, desde a identificação das melhores fontes de informação e dos envolvidos na tomada de decisão, até as técnicas e ferramentas usadas no levantamento de requisitos, no projeto da solução e nos testes de aplicação.

Compre agora e leia

Table of Contents